

37th European Conference on Object-Oriented Programming

ECOOP 2023, July 17–21, 2023, Seattle, Washington,
United States

Edited by

Karim Ali

Guido Salvaneschi



Editors

Karim Ali 

University of Alberta, Canada
karim.ali@ualberta.ca

Guido Salvaneschi 

University of St. Gallen, Switzerland
guido.salvaneschi@unisg.ch

ACM Classification 2012

Software and its engineering

ISBN 978-3-95977-281-5

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-281-5>.

Publication date

July, 2023

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): <https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2023.0

ISBN 978-3-95977-281-5

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

LIPICs – Leibniz International Proceedings in Informatics

LIPICs is a series of high-quality conference proceedings across all fields in informatics. LIPICs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB and Nanyang Technological University, SG)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Message from the Program Chairs <i>Karim Ali and Guido Salvaneschi</i>	0:ix
Message from the Artifact Evaluation Chairs <i>Stefan Winter and Hernan Luis Ponce de Leon</i>	0:xi
Foreword by the President of AITO <i>Eric Jul</i>	0:xiii
List of Authors	0:xv

Regular Papers

Designing Asynchronous Multiparty Protocols with Crash-Stop Failures <i>Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou</i>	1:1–1:30
Nested Pure Operation-Based CRDTs <i>Jim Bauwens and Elisa Gonzalez Boix</i>	2:1–2:26
Multi-Graded Featherweight Java <i>Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca</i>	3:1–3:27
Hoogle \star : Constants and λ -abstractions in Petri-net-based Synthesis using Symbolic Execution <i>Henrique Botelho Guerra, João F. Ferreira, and João Costa Seco</i>	4:1–4:28
Modular Abstract Definitional Interpreters for WebAssembly <i>Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen</i>	5:1–5:28
Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols <i>David Castro-Perez and Nobuko Yoshida</i>	6:1–6:30
Modular Compilation for Higher-Order Functional Choreographies <i>Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti</i>	7:1–7:37
Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It... <i>Jan de Muijnck-Hughes and Wim Vanderbauwhede</i>	8:1–8:28
VeriFx: Correct Replicated Data Types for the Masses <i>Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix</i>	9:1–9:45
On Leveraging Tests to Infer Nullable Annotations <i>Jens Dietrich, David J. Pearce, and Mahin Chandramohan</i>	10:1–10:25
super -Charging Object-Oriented Programming Through Precise Typing of Open Recursion <i>Andong Fan and Lionel Parreaux</i>	11:1–11:28



LoRe: A Programming Model for Verifiably Safe Local-First Software (Extended Abstract) <i>Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini</i>	12:1–12:15
Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises <i>Feiyang Jin, Lechen Yu, Tiago Cogumbreiro, Jun Shirako, and Vivek Sarkar</i>	13:1–13:30
Algebraic Replicated Data Types: Programming Secure Local-First Software <i>Christian Kuessner, Ragnar Mogk, Anna-Katharina Wickert, and Mira Mezini</i> ...	14:1–14:33
Behavioural Types for Local-First Software <i>Roland Kuhn, Hernán Melgratti, and Emilio Tuosto</i>	15:1–15:28
Constraint Based Compiler Optimization for Energy Harvesting Applications <i>Yannan Li and Chao Wang</i>	16:1–16:29
Restrictable Variants: A Simple and Practical Alternative to Extensible Variants <i>Magnus Madsen, Jonathan Lindegaard Starup, and Matthew Lutze</i>	17:1–17:27
Programming with Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism <i>Magnus Madsen and Jaco van de Pol</i>	18:1–18:27
Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding <i>Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner</i>	19:1–19:27
Morpheus: Automated Safety Verification of Data-Dependent Parser Combinator Programs <i>Ashish Mishra and Suresh Jagannathan</i>	20:1–20:27
Automata Learning with an Incomplete Teacher <i>Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva</i>	21:1–21:30
Modular Verification of State-Based CRDTs in Separation Logic <i>Abel Nieto, Arnaud Daby-Seesaram, Léon Gondelman, Amin Timany, and Lars Birkedal</i>	22:1–22:27
Information Flow Analysis for Detecting Non-Determinism in Blockchain <i>Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto</i>	23:1–23:25
Toward Tool-Independent Summaries for Symbolic Execution <i>Frederico Ramos, Nuno Sabino, Pedro Adão, David A. Naumann, and José Fragoso Santos</i>	24:1–24:29
A Direct-Style Effect Notation for Sequential and Parallel Programs <i>David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini</i>	25:1–25:22
SINATRA: Stateful Instantaneous Updates for Commercial Browsers Through Multi-Version eXecution <i>Ugnius Rumsevicius, Siddhanth Venkateshwaran, Ellen Kidane, and Luís Pina</i> ...	26:1–26:29

An Efficient Vectorized Hash Table for Batch Computations <i>Hesam Shahrokhi and Amir Shaikhha</i>	27:1–27:27
Hinted Dictionaries: Efficient Functional Ordered Sets and Maps <i>Amir Shaikhha, Mahdi Ghorbani, and Hesam Shahrokhi</i>	28:1–28:30
Semantics for Noninterference with Interaction Trees <i>Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic</i>	29:1–29:29
Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification <i>Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott</i>	30:1–30:26
Breaking the Negative Cycle: Exploring the Design Space of Stratification for First-Class Datalog Constraints <i>Jonathan Lindegaard Starup, Magnus Madsen, and Ondřej Lhoták</i>	31:1–31:28
Asynchronous Multiparty Session Type Implementability is Decidable – Lessons Learned from Message Sequence Charts <i>Felix Stutz</i>	32:1–32:31
ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs <i>Felix Suchert, Lisza Zeidler, Jeronimo Castrillon, and Sebastian Ertel</i>	33:1–33:39
Dependent Merges and First-Class Environments <i>Jinhao Tan and Bruno C. d. S. Oliveira</i>	34:1–34:32
Synthesis-Aided Crash Consistency for Storage Systems <i>Jacob Van Geffen, Xi Wang, Emina Torlak, and James Bornholt</i>	35:1–35:26
Synthesizing Conjunctive Queries for Code Search <i>Chengpeng Wang, Peisen Yao, Wensheng Tang, Gang Fan, and Charles Zhang</i> ...	36:1–36:30
Do Machine Learning Models Produce TypeScript Types That Type Check? <i>Ming-Ho Yee and Arjun Guha</i>	37:1–37:28

Experience Papers

Building Code Transpilers for Domain-Specific Languages Using Program Synthesis <i>Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung</i>	38:1–38:30
Rust for Morello: Always-On Memory Safety, Even in Unsafe Code <i>Sarah Harris, Simon Cooksey, Michael Vollmer, and Mark Batty</i>	39:1–39:27
On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage <i>João Mota, Marco Giunti, and António Ravara</i>	40:1–40:29

Pearls/Brave New Ideas

The Dolorem Pattern: Growing a Language Through Compile-Time Function Execution <i>Simon Henniger and Nada Amin</i>	41:1–41:27
--	------------

Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types	
<i>Sung-Shik Jongmans and Francisco Ferreira</i>	42:1–42:30
On the Rise of Modern Software Documentation	
<i>Marco Raglianti, Csaba Nagy, Roberto Minelli, Bin Lin, and Michele Lanza</i>	43:1–43:24
Python Type Hints Are Turing Complete	
<i>Ori Roth</i>	44:1–44:15

■ Message from the Program Chairs

Started in 1987, ECOOP is Europe's oldest programming conference, welcoming papers on all practical and theoretical investigations of programming languages, systems and environment providing innovative solutions to real problems as well as evaluations of existing solutions. Papers were submitted to one of four categories: *Research* for papers that advance the state of the art in programming; *Reproduction* for empirical evaluations that reconstructs a published experiment in a different context in order to validate the results of that earlier work; *Experience* for applications of known techniques in practice; and *Pearl* for papers that either explain a known idea in an elegant way or unconventional papers introducing ideas that may take some time to substantiate. ECOOP is a selective venue, with acceptance, by tradition, capped at 25% of all submissions and re-submissions. The chairs thank the Program Committee: B. Hermann, C. Omar, E. Söderberg, G. Agha, R. Baghdadi, S. Chiba, A. Craik, W. De Meuter, A. F. Donaldson, S. J. Gay, J. Gibbons, T. Hosking, A. Igarashi, M. Luján, A. Milanova, A. Möller, K. Ostermann, T. Petricek, A. Potanin, T. Schrijvers, M. Serrano, T. Sotiropoulos, P. Thiemann, E. Tosch, V. T. Vasconcelos, Y. Wang, S. Wehr, T. Wrigstad, and C. Zhang. This year, we continued a number of innovations that were first introduced in 2022:

- **Multiple rounds.** ECOOP has two main rounds of submissions per year (Dec 1 and Mar 1). Each round supports both minor and major revisions. Major revisions are handled in the next round (either the same year or the next) by the same reviewers.
- **No format or length restrictions.** In order to reduce friction for authors, papers can come in any format and at any length. This applies to submissions. Final versions must abide by the publisher's requirements.
- **Artifacts and Papers together.** Every submitted paper can be accompanied with an artifact, submitted a few days after the paper. Both submissions are evaluated in parallel by overlapping committees as members of the artifact evaluation committee were invited to served on the conference review committee.
- **Journal First/Last.** Papers can be submitted either one of three associated journals and be invited to present at the meeting. Furthermore, some accepted papers can be forward to journals.

Overall, we found that most of these innovations to have worked well. We hope that future chairs will continue to experiment with more, and perhaps, different innovations that will enrich the ECOOP community further.

Karim Ali

Program Committee Co-chair

University of Alberta

Guido Salvaneschi

Program Committee Co-chair

University of St. Gallen

■ Message from the Artifact Evaluation Chairs

ECOOP has a long-standing tradition of offering artifact evaluation dating back to 2013. Following the process introduced in 2022, the artifact evaluation involved every single paper submission to ECOOP 2023, rather than just accepted papers, and happened in parallel with the paper review process. Besides providing feedback on the artifacts irrespective of paper acceptance, evaluation results were made available to the technical PC. Artifact submissions could, thus, provide more insights on the technical contributions described in the papers and help to improve the overall review process.

To handle the high review load that such a process entails, we recruited a large artifact evaluation committee that included a total of 51 artifact reviewers. The submission deadlines for artifacts were just 10 days after the paper deadlines for both submission rounds. We received a total of 45 submissions (20 for R1 and 25 for R2). After a kick-the-tires review and author response phase, during which authors had the opportunity to clarify or address technical issues with their submissions, each submitted artifact was reviewed by three committee members, leading to an overall review load of around 3 artifact reviews per committee member.

Following the positive experience with adopting ACM's artifact badges for ECOOP 2021 and 2022, we adopted the same badging policies for ECOOP 2023. The artifact evaluation committee positively evaluated 38 submissions (15/23 for R1/R2) as *functional* or *reusable*, out of which 22 belong to papers to appear in the technical program of ECOOP 2023. 4 submitted artifacts that did not pass the bar for the *functional* and *reusable* badges in R1 were found eligible for the *available* badge, 2 of which are associated with papers accepted for presentation at ECOOP 2023.

In order to streamline the artifact review process and to decouple artifact from paper review aspects, we asked authors to submit documentation of explicit *claims* in a pre-specified format that the artifact evaluation committee checked the artifacts against. At the same time, the PC could assess the *importance of these claims* for the submitted papers as a frame of reference for the strength of support for the paper that an artifact can provide. This separation greatly facilitated the artifact evaluation committee's discussions regarding which badges to award.

The smooth and thorough artifact evaluation process would have not been possible without the members of the artifact evaluation committee, who handled the artifact review workload and contributed to the technical PC discussions with great dedication. We would like to thank them for their valuable work and the inspiring discussions. We would also like to thank the ECOOP 2023 program committee chairs Karim Ali and Guido Salvaneschi for the pleasant and productive interactions over the coordination of the paper and artifact review processes.

Hernán Ponce de León

Artifact Evaluation Co-chair

Huawei Dresden Research Center

Stefan Winter

Artifact Evaluation Co-chair

Ludwig-Maximilians-Universität München

■ Foreword by the President of AITO


Welcome to ECOOP 2023, which this time will be held in the “well-known European city of Seattle”. Why outside Europe? Well, ECOOP traditionally has had many contributors and participants from other parts of the world and so ECOOP every once in a while has been held outside Europe. In 1990, ECOOP was co-located with OOPSLA in Ottawa, Canada, and in 2012, ECOOP was co-located with PLDI, LCTES, and ISMM in Beijing, China. This year, we are co-locating with ISSTA at the University of Washington main campus beautifully located by Lake Washington and with splendid views of the Cascade Mountains and Mount Rainier. The ECOOP 2023 team along with the ISSTA team has done a great job of putting together a great program for the conferences – a huge thanks to them and to all others that have contributed.

I am looking forward to two excellent conferences with lots of great papers, personal interaction, excellent keynotes, including talks by the two 2023 Dahl-Nygaard Prize Winners. Enjoy the conference, and Seattle.

Eric Jul
AITO President




■ List of Authors

Pedro Adão  (24)
Instituto Superior Técnico,
University of Lisbon, Portugal;
Institute of Telecommunications,
Campus de Santiago, Aveiro, Portugal

Nada Amin (41)
Harvard University, Cambridge, MA, USA


Vincenzo Arceri  (23)
University of Parma, Italy


Adam D. Barwell  (1)
University of St. Andrews, UK;
University of Oxford, UK

Mark Batty (39)
University of Kent, Canterbury, UK


Jim Bauwens (2)
Software Languages Lab,
Vrije Universiteit Brussel, Belgium


Sahil Bhatia (38)
University of California, Berkeley, CA, USA

Riccardo Bianchini  (3)
DIBRIS, University of Genova, Italy

Annette Bieniusa  (12)
University of Kaiserslautern-Landau, Germany

Lars Birkedal  (22)
Aarhus University, Denmark

James Bornholt  (35)
The University of Texas at Austin, TX, USA;
Amazon Web Services, Seattle, WA, USA

Henrique Botelho Guerra  (4)
INESC-ID and IST, University of Lisbon,
Portugal

Katharina Brandl (5)
Johannes Gutenberg-Universität Mainz,
Germany

Timon Böhler  (25)
Technische Universität Darmstadt, Germany

Jeronimo Castrillon  (33)
TU Dresden, Germany

David Castro-Perez  (6)
University of Kent, UK

Ethan Cecchetti  (29)
University of Maryland, College Park, MD, USA;
University of Wisconsin – Madison, WI, USA


Mahin Chandramohan (10)
Oracle Labs, Brisbane, Australia

Alvin Cheung (38)
University of California, Berkeley, CA, USA


Tiago Cogumbreiro (13)
College of Science and Mathematics, University
of Massachusetts Boston, MA, USA


Simon Cooksey  (39)
University of Kent, Canterbury, UK


Agostino Cortesi  (23)
Ca' Foscari University of Venice, Italy

João Costa Seco  (4)
NOVA LINCS, NOVA School of Science and
Technology, Caparica, Portugal

Caroline Cronjäger (19)
Ruhr-Universität Bochum, Germany


Luís Cruz-Filipe  (7)
Department of Mathematics and Computer
Science, University of Southern Denmark,
Odense, Denmark

Arnaud Daby-Seesaram  (22)
ENS Paris-Saclay, France


Francesco Dagnino  (3)
DIBRIS, University of Genova, Italy

Jan de Muijnck-Hughes  (8)
University of Glasgow, UK


Kevin De Porre  (9)
Vrije Universiteit Brussel, Belgium

Jens Dietrich  (10)
Victoria University of Wellington, New Zealand

Sebastian Erdweg (5)
Johannes Gutenberg-Universität Mainz,
Germany

Sebastian Ertel  (33)
Barkhausen Institut, Dresden, Germany

Andong Fan  (11)
The Hong Kong University of Science and
Technology (HKUST), Hong Kong, China

Gang Fan  (36)
Ant Group, Shenzhen, China


Pietro Ferrara  (23)
Ca' Foscari University of Venice, Italy


- Carla Ferreira  (9)
NOVA School of Science and Technology,
Caparica, Portugal
- Francisco Ferreira (42)
Department of Computer Science, Royal
Holloway, University of London, UK
- João F. Ferreira  (4)
INESC-ID and IST, University of Lisbon,
Portugal
- Nate Foster  (21)
Cornell University, Ithaca, NY, USA
- José Fragoso Santos  (24)
Instituto Superior Técnico,
University of Lisbon, Portugal;
INESC-ID Lisbon, Portugal
- Philippa Gardner (19)
Imperial College London, UK
- Mahdi Ghorbani (28)
University of Edinburgh, UK
- Paola Giannini  (3)
DiSSTE, University of Eastern Piedmont,
Vercelli, Italy
- Marco Giunti  (40)
NOVA LINCS and NOVA School of Science and
Technology, Caparica, Portugal;
School of Computing, Engineering & Digital
Technologies, Teesside University,
Middlesbrough, UK
- Léon Gondelman  (22)
Aarhus University, Denmark
- Elisa Gonzalez Boix (2, 9)
Software Languages Lab,
Vrije Universiteit Brussel, Belgium
- Eva Graversen  (7)
Department of Mathematics and Computer
Science, University of Southern Denmark,
Odense, Denmark
- Arjun Guha  (37)
Northeastern University, Boston, MA, USA;
Roblox Research, San Mateo, CA, USA
- Julian Haas  (12)
Technische Universität Darmstadt, Germany
- Nils Hansen (5)
Johannes Gutenberg-Universität Mainz,
Germany
- Sarah Harris (39)
University of Kent, Canterbury, UK
- Paul He  (29)
University of Pennsylvania,
Philadelphia, PA, USA
- Simon Henniger (41)
Technische Universität München, Germany
- Andrew K. Hirsch  (29)
State University of New York at Buffalo,
NY, USA
- Ping Hou  (1)
University of Oxford, UK
- Suresh Jagannathan  (20)
Department of Computer Science,
Purdue University, West Lafayette, IN, USA
- Feiyang Jin (13)
College of Computing, Georgia Institute of
Technology, Atlanta, GA, USA
- Sung-Shik Jongmans (42)
Department of Computer Science, Open
University, Heerlen, The Netherlands;
Centrum Wiskunde & Informatica (CWI),
NWO-I, Amsterdam, The Netherlands
- Sven Keidel (5)
TU Darmstadt, Germany
- Ellen Kidane (26)
University of Illinois at Chicago, IL, USA
- Caleb Koch (21)
Stanford University, CA, USA
- Sumer Kohli (38)
University of California, Berkeley, CA, USA
- Christian Kuessner  (14)
Technische Universität Darmstadt, Germany
- Roland Kuhn  (15)
Actyx AG, Kassel, Germany
- Michele Lanza  (43)
REVEAL @ Software Institute – USI, Lugano,
Switzerland
- Ondřej Lhoták (31)
David R. Cheriton School of Computer Science,
University of Waterloo, Canada
- Yannan Li (16)
University of Southern California, Los Angeles,
CA, USA
- Bin Lin  (43)
Radboud University, Nijmegen, The Netherlands

- Lovro Lugović  (7)
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
- Matthew Lutze  (17)
Department of Computer Science, Aarhus University, Denmark
- Andreas Lööw (19)
Imperial College London, UK
- Magnus Madsen  (17, 18, 31)
Department of Computer Science, Aarhus University, Denmark
- Petar Maksimović (19)
Imperial College London, UK;
Runtime Verification Inc., Urbana, IL, USA
- Hernán Melgratti  (15)
University of Buenos Aires, Argentina;
Conicet, Buenos Aires, Argentina
- Mira Mezini  (12, 14, 25)
hessian.AI, Darmstadt, Germany;
Technische Universität Darmstadt, Germany
- Roberto Minelli  (43)
REVEAL @ Software Institute – USI, Lugano, Switzerland
- Ashish Mishra  (20)
Department of Computer Science,
Purdue University, West Lafayette, IN, USA
- Mark Moeller  (21)
Cornell University, Ithaca, NY, USA
- Ragnar Mogk  (12, 14)
Technische Universität Darmstadt, Germany
- Fabrizio Montesi  (7)
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
- João Mota  (40)
NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal
- Csaba Nagy  (43)
REVEAL @ Software Institute – USI, Lugano, Switzerland
- David A. Naumann  (24)
Stevens Institute of Technology,
Hoboken, NJ, USA
- Luca Negrini  (23)
Corvallis Srl, Padova, Italy
- Abel Nieto  (22)
Aarhus University, Denmark
- Bruno C. d. S. Oliveira (34)
The University of Hong Kong, China
- Luca Olivieri  (23)
University of Verona, Italy;
Corvallis Srl, Padova, Italy
- Lionel Parreaux  (11)
The Hong Kong University of Science and Technology (HKUST), Hong Kong, China
- David J. Pearce  (10)
ConsenSys, Wellington, New Zealand
- Marco Peressotti  (7)
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
- Luís Pina  (26)
University of Illinois at Chicago, IL, USA
- Marco Raglianti  (43)
REVEAL @ Software Institute – USI, Lugano, Switzerland
- Frederico Ramos  (24)
Instituto Superior Técnico, University of Lisbon, Portugal; INESC-ID Lisbon, Portugal
- António Ravara  (40)
NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal
- David Richter  (25)
Technische Universität Darmstadt, Germany
- Ori Roth  (44)
Department of Computer Science,
Technion, Haifa, Israel
- Ugnius Rumsevicius (26)
University of Illinois at Chicago, IL, USA
- Nuno Sabino  (24)
Instituto Superior Técnico,
University of Lisbon, Portugal;
Carnegie Mellon University,
Pittsburgh, PA, USA;
Institute of Telecommunications,
Campus de Santiago, Aveiro, Portugal
- Vivek Sarkar (13)
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
- Ryan Scott (30)
Galois, Inc., Portland, OR, USA

- Sanjit A. Seshia (38)
University of California, Berkeley, CA, USA
- Hesam Shahrokhi  (27, 28)
University of Edinburgh, UK
- Amir Shaikhha  (27, 28)
University of Edinburgh, UK
- Jun Shirako (13)
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
- Alexandra Silva  (21)
Cornell University, Ithaca, NY, USA
- Lucas Silver (29, 30)
University of Pennsylvania,
Philadelphia, PA, USA
- Alaia Solko-Breslin  (21)
University of Pennsylvania,
Philadelphia, PA, USA
- Fausto Spoto  (23)
University of Verona, Italy
- Jonathan Lindegaard Starup  (17, 31)
Department of Computer Science,
Aarhus University, Denmark
- Felix Stutz  (32)
MPI-SWS, Kaiserslautern, Germany
- Felix Suchert  (33)
TU Dresden, Germany
- Julian Sutherland (19)
Nethermind, London, UK
- Fabio Tagliaferro  (23)
CYS4 Srl, Florence, Italy
- Jinhao Tan (34)
The University of Hong Kong, China
- Wensheng Tang  (36)
The Hong Kong University of Science and Technology, China
- Amin Timany  (22)
Aarhus University, Denmark
- Emina Torlak (35)
University of Washington, Seattle, WA, USA;
Amazon Web Services, Seattle, WA, USA
- Emilio Tuosto  (15)
Gran Sasso Science Institute, L'Aquila, Italy
- Jaco van de Pol  (18)
Department of Computer Science,
Aarhus University, Denmark
- Jacob Van Geffen  (35)
University of Washington, Seattle, WA, USA
- Wim Vanderbauwhede  (8)
University of Glasgow, UK
- Siddhanth Venkateshwaran (26)
University of Illinois at Chicago, IL, USA
- Michael Vollmer  (39)
University of Kent, Canterbury, UK
- Chao Wang (16)
University of Southern California,
Los Angeles, CA, USA
- Chengpeng Wang  (36)
The Hong Kong University of Science and Technology, China
- Xi Wang (35)
University of Washington, Seattle, WA, USA;
Amazon Web Services, Seattle, WA, USA
- Pascal Weisenburger  (25)
Universität St. Gallen, Switzerland
- Eddy Westbrook (30)
Galois, Inc., Portland, OR, USA
- Anna-Katharina Wickert  (14)
Technische Universität Darmstadt, Germany
- Thomas Wiener (21)
Cornell University, Ithaca, NY, USA
- Matthew Yacavone (30)
Galois, Inc., Portland, OR, USA
- Elena Yanakieva  (12)
University of Kaiserslautern-Landau, Germany
- Peisen Yao  (36)
Zhejiang University, Hangzhou, China
- Ming-Ho Yee  (37)
Northeastern University, Boston, MA, USA
- Nobuko Yoshida  (1, 6)
University of Oxford, UK
- Lechen Yu (13)
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
- Steve Zdancewic  (29)
University of Pennsylvania,
Philadelphia, PA, USA
- Lisza Zeidler (33)
Barkhausen Institut, Dresden, Germany

Charles Zhang  (36)
The Hong Kong University of Science and
Technology, China

Fangyi Zhou  (1)
Imperial College London, UK;
University of Oxford, UK

Elena Zucca  (3)
DIBRIS, University of Genova, Italy

Designing Asynchronous Multiparty Protocols with Crash-Stop Failures

Adam D. Barwell  



University of St. Andrews, UK
University of Oxford, UK

Ping Hou  

University of Oxford, UK

Nobuko Yoshida  

University of Oxford, UK

Fangyi Zhou  

Imperial College London, UK
University of Oxford, UK

Abstract

Session types provide a typing discipline for message-passing systems. However, most session type approaches assume an ideal world: one in which everything is reliable and without failures. Yet this is in stark contrast with distributed systems in the real world. To address this limitation, we introduce TEATRINO, a code generation toolchain that utilises asynchronous *multiparty session types* (MPST) with *crash-stop* semantics to support failure handling protocols.

We augment asynchronous MPST and processes with *crash handling* branches. Our approach requires no user-level syntax extensions for global types and features a formalisation of global semantics, which captures complex behaviours induced by crashed/crash handling processes. The sound and complete correspondence between global and local type semantics guarantees deadlock-freedom, protocol conformance, and liveness of typed processes in the presence of crashes.

Our theory is implemented in the toolchain TEATRINO, which provides *correctness by construction*. TEATRINO extends the SCRIBBLE multiparty protocol language to generate protocol-conforming SCALA code, using the EFFPI concurrent programming library. We extend both SCRIBBLE and EFFPI to support *crash-stop* behaviour. We demonstrate the feasibility of our methodology and evaluate TEATRINO with examples extended from both session type and distributed systems literature.

2012 ACM Subject Classification Software and its engineering → Source code generation; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi; Theory of computation → Distributed computing models

Keywords and phrases Session Types, Concurrency, Failure Handling, Code Generation, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.1

Related Version *Full Version*: <https://arxiv.org/abs/2305.06238>

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*:

<https://doi.org/10.4230/DARTS.9.2.9>

Software (ECOOP 2023 Artifact Evaluation approved artifact):

<http://doi.org/10.5281/zenodo.7714133>

Software (Source Code): <https://github.com/adbarwell/ECOOP23-Artifact>

archived at `swh:1:dir:e680ab478b62aab45610b0ef9f6de9d0fbe20ad2`

Funding Work supported by: EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We thank the anonymous reviewers for their useful comments and suggestions. We thank Jia Qing Lim for his contribution to the EFFPI extension. We thank Alceste Scalas for useful discussions and advice in the development of this paper and for his assistance with EFFPI.



© Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 1; pp. 1:1–1:30



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Background. As distributed programming grows increasingly prevalent, significant research effort has been devoted to improve the reliability of distributed systems. A key aspect of this research focuses on studying *unreliability* (or, more specifically, failures). Modelling unreliability and failures enables a distributed system to be designed to be more tolerant of failures, and thus more resilient.

In pursuit of methods to achieve safety in distributed communication systems, *session types* [19] provide a lightweight, type system–based approach to message-passing concurrency. In particular, *Multiparty Session Types* (MPST) [20] facilitate the specification and verification of communication between message-passing processes in concurrent and distributed systems. The typing discipline prevents common communication-based errors, e.g. deadlocks and communication mismatches [21, 37]. On the practical side, MPST have been implemented in various mainstream programming languages [7, 10, 11, 22, 24, 25, 28, 30], which facilitates their applications in real-world programs.

Nevertheless, the challenge to account for unreliability and failures persists for session types: most session type systems assume that both participants and message transmissions are *reliable* without failures. In a real-world setting, however, participants may crash, communications channels may fail, and messages may be lost. The lack of failure modelling in session type theories creates a barrier to their applications to large-scale distributed systems.

Recent works [3, 26, 27, 33, 42] close the gap of failure modelling in session types with various techniques. [42] introduces *failure suspicion*, where a participant may suspect their communication partner has failed, and act accordingly. [33] introduces *reliability annotations* at type level, and fall back to a given *default* value in case of failures. [26] proposes a framework of *affine* multiparty session types, where a session can terminate prematurely, e.g. in case of failures. [3] integrates *crash-stop failures*, where a generalised type system validates safety and liveness properties with model checking; [27] takes a similar approach, modelling more kinds of failures in a session type system, e.g. message losses, reordering, and delays.

While steady advancements are made on the theoretical side, the implementations of those enhanced session type theories seem to lag behind. Barring the approaches in [26, 42], the aforementioned approaches [3, 27, 33] do not provide session type API support for programming languages.¹ To bring the benefits of the theoretical developments into real-world distributed programming, a gap remains to be filled on the implementation side.

This Paper. We introduce a *top-down* methodology for designing asynchronous multiparty protocols with crash-stop failures:

- (1) We use an extended asynchronous MPST theory, which models *crash-stop* failures, and show that the usual session type guarantees remain valid, i.e. communication safety, deadlock-freedom, and liveness;
- (2) We present a toolchain for implementing asynchronous multiparty protocols, under our new asynchronous MPST theory, in SCALA, using the EFFPI concurrency library [38].

The top-down design methodology comes from the original MPST theory [20], where the design of multiparty protocols begins with a given *global* type (top), and implementations rely on *local* types (bottom) obtained from the global type. The global and local types reflect the

¹ [3] provides a prototype implementation, utilising the mCRL2 model checker [5], for verifying type-level properties, instead of a library for general use.

global and local communication behaviours respectively. Well-typed implementations that conform to a global type are guaranteed to be *correct by construction*, enjoying full guarantees (safety, deadlock-freedom, liveness) from the theory. This remains the predominant approach for implementing MPST theories, and is also followed by some aforementioned systems [26, 42].

We model *crash-stop* failures [6, §2.2], i.e. a process may fail arbitrarily and cease to interact with others. This model is simple and expressive, and has been adopted by other approaches [3, 27]. Using global types in our design for handling failures in multiparty protocols presents two distinct advantages:

- (1) global types provide a simple, high-level means to both specify a protocol abstractly and automatically derive local types; and,
- (2) desirable behavioural properties such as communication safety, deadlock-freedom, and liveness are guaranteed by construction.

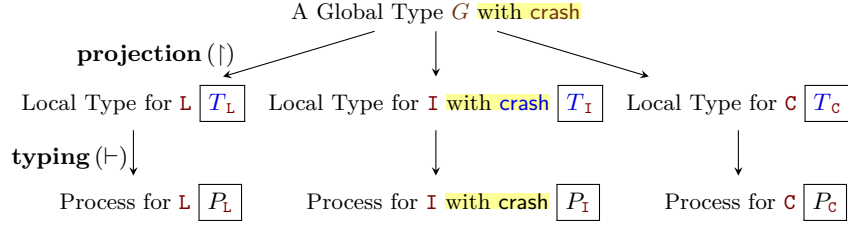
In contrast to the *synchronous* semantics in [3], we model an *asynchronous* semantics, where messages are buffered whilst in transit. We focus on asynchronous systems since most communication in the real distributed world is asynchronous. In [27], although the authors develop a generic typing system incorporating asynchronous semantics, their approach results in the type-level properties becoming undecidable [27, §4.4]. With global types, we restore the decidability at a minor cost to expressivity.

To address the gap on the practical side, we present a code generator toolchain, TEATRINO, to implement our MPST theory. Our toolchain takes an asynchronous multiparty protocol as input, using the protocol description language SCRIBBLE [43], and generates SCALA code using the EFFPI [38] concurrency library as output.

The SCRIBBLE Language [43] is designed for describing multiparty communication protocols, and is closely connected to MPST theory (cf. [31]). This language enables a programmatic approach for expressing global types and designing multiparty protocols. The EFFPI concurrency library [38] offers an embedded Domain Specific Language (DSL) that provides a simple actor-based API. The library offers both type-level and value-level constructs for processes and channels. Notably, the type-level constructs reflect the behaviour of programs (i.e. processes) and can be used as specifications. Our code generation technique, as well as the EFFPI library itself, utilises the type system features introduced in SCALA 3, including match types and dependent function types, to encode local types in EFFPI. This approach enables us to specify and verify program behaviour at the type level, resulting in a more powerful and flexible method for handling concurrency.

By extending SCRIBBLE and EFFPI to support *crash detection and handling*, our toolchain TEATRINO provides a lightweight way for developers to take advantage of our theory, bridging the gap on the practical side. We evaluate the expressivity and feasibility of TEATRINO with examples incorporating crash handling behaviour, extended from session type literature.

Outline. We begin with an overview of our methodology in § 2. We introduce an asynchronous multiparty session calculus in § 3 with semantics of crashing and crash handling. We introduce an extended theory of asynchronous multiparty session types with semantic modelling of crash-stop failures in § 4. We present a typing system for the multiparty session calculus in § 5. We introduce TEATRINO, a code generation toolchain that implements our theory in § 6, demonstrating how our approach is applied in the SCALA programming language. We evaluate our toolchain with examples from both session type and distributed systems literature in § 7. We discuss related work in § 8 and conclude in § 9. Full proofs, auxiliary material, and more details of TEATRINO can be found in the full version of the paper [2]. Additionally, our toolchain and examples used in our evaluation are available on [GitHub](#).



■ **Figure 1** Top-down View of MPST with Crash.

2 Overview

In this section, we give an overview of our methodology for designing asynchronous multiparty protocols with crash-stop failures, and demonstrate our code generation toolchain, TEATRINO.

Asynchronous Multiparty Protocols with Crash-Stop Failures. We follow a standard top-down design approach enabling *correctness by construction*, but enrich asynchronous MPST with crash-stop semantics. As depicted in Fig. 1, we formalise (asynchronous) multiparty protocols with crash-stop failures as global types with *crash handling branches* (*crash*). These are projected into local types, which may similarly contain crash handling branches (*crash*). The projected local types are then used to type-check processes (also with crash handling branches (*crash*)) that are written in a session calculus. As an example, we consider a simple *distributed logging* scenario, which is inspired by the logging-management protocol [26], but extended with a third participant.

The Simpler Logging protocol consists of a *logger* (**L**) that controls the logging services, an *interface* (**I**) that provides communications between logger and client, and a *client* (**C**) that requires logging services via interface. Initially, **L** sends a heartbeat message *trigger* to **I**. Then **C** sends a command to **L** to read the logs (*read*). When a *read* request is sent, it is forwarded to **L**, and **L** responds with a *report*, which is then forwarded onto **C**. Assuming all participants (logger, interface, and client) are reliable, i.e. without any failures or crashes, this logging behaviour can be represented by the *global type* G_0 :

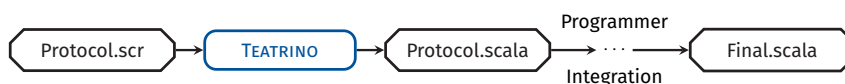
$$G_0 = \mathbf{L} \rightarrow \mathbf{I} : \text{trigger} . \mathbf{C} \rightarrow \mathbf{I} : \text{read} . \mathbf{I} \rightarrow \mathbf{L} : \text{read} . \mathbf{L} \rightarrow \mathbf{I} : \text{report}(\text{log}) . \mathbf{I} \rightarrow \mathbf{C} : \text{report}(\text{log}) . \text{end} \quad (1)$$

Here, G_0 is a specification of the Simpler Logging protocol between multiple roles from a global perspective.

In the real distributed world, all participants in the Simpler Logging system may fail. Ergo, we need to model protocols with failures or crashes and handling behaviour, e.g. should the client fail after the logging has started, the interface will inform the logger to stop and exit. We follow [6, §2.2] to model a *crash-stop* semantics, where we assume that roles can crash *at any time* unless assumed *reliable* (never crash). For simplicity, we assume **I** and **L** to be reliable. The above logging behaviour, incorporating crash-stop failures, can be represented by extending G_0 with a branch handling a crash of **C**:

$$G = \mathbf{L} \rightarrow \mathbf{I} : \text{trigger} . \mathbf{C} \rightarrow \mathbf{I} : \left\{ \begin{array}{l} \text{read} . \mathbf{I} \rightarrow \mathbf{L} : \text{read} . \mathbf{L} \rightarrow \mathbf{I} : \text{report}(\text{log}) . \mathbf{I} \rightarrow \mathbf{C} : \text{report}(\text{log}) . \text{end} \\ \text{crash} . \mathbf{I} \rightarrow \mathbf{L} : \text{fatal} . \text{end} \end{array} \right\} \quad (2)$$

We model crash detection on receiving roles: when **I** is waiting to receive a message from **C**, the receiving role **I** is able to detect whether **C** has crashed. Since crashes are detected only by the receiving role, we do not require a crash handling branch on the communication step between **I** and **C** – nor do we require them on any interaction between **L** and **I** (since we are assuming that **L** and **I** are reliable).



■ **Figure 2** Workflow of TEATRINO.

Following the MPST top-down methodology, a global type is then *projected* onto *local types*, which describe communications from the perspective of a single role. In our unreliable Simpler Logging example, G is projected onto three local types (one for each role C, L, I):

$$T_C = I \oplus \text{read}.I \& \text{report}(\log).\text{end} \quad T_L = I \oplus \text{trigger}.I \& \left\{ \begin{array}{l} \text{read}.I \oplus \text{report}(\log).\text{end} \\ \text{fatal}.\text{end} \end{array} \right\}$$

$$T_I = L \& \text{trigger}.C \& \left\{ \begin{array}{l} \text{read}.L \oplus \text{read}.L \& \text{report}(\log).C \oplus \text{report}(\log).\text{end} \\ \text{crash}.L \oplus \text{fatal}.\text{end} \end{array} \right\}$$

Here, T_I states that I first receives a trigger message from L ; then I either expects a **read** request from C , or detects the crash of C and handles it (in **crash**) by sending the **fatal** message to notify L . We add additional crash modelling and introduce a **stop** type for crashed endpoints. We show the operational correspondence between global and local type semantics, and demonstrate that a projectable global type always produces a safe, deadlock-free, and live typing context.

The next step in this top-down methodology is to use local types to type-check processes P_i executed by role p_i in our session calculus. For example, T_I can be used to type check I that executes the process:

$$L? \text{trigger} . \sum \left\{ \begin{array}{l} C? \text{read}.L! \text{read}.L? \text{report}(x).C! \text{report}(x).0 \\ C? \text{crash}.L! \text{fatal}.0 \end{array} \right\}$$

In our operational semantics (§3), we allow active processes executed by unreliable roles to crash *arbitrarily*. Therefore, the role executing the crashed process also crashes, and is assigned the local type **stop**. To ensure that a communicating process is type-safe even in presence of crashes, we require that its typing context satisfies a *safety property* accounting for possible crashes (Def. 13), which is preserved by projection. Additional semantics surrounding crashes adds subtleties even in standard results. We prove subject reduction and session fidelity results accounting for crashes and sets of reliable roles.

Code Generation Toolchain: Teatrino. To complement the theory, we present a code generation toolchain, TEATRINO, that generates protocol-conforming SCALA code from a multiparty protocol. We show the workflow diagram of our toolchain in Fig. 2. TEATRINO takes a SCRIBBLE protocol (Protocol.scr) and generates executable code (Protocol.scala) conforming to that protocol, which the programmer can integrate with existing code (Final.scala).

TEATRINO implements our session type theory to handle global types expressed using the SCRIBBLE protocol description language [43], a programmer-friendly way for describing multiparty protocols. We extend the syntax of SCRIBBLE slightly to include constructs for **crash** recovery branches and reliable roles.

The generated SCALA code utilises the EFFPI concurrency library [38]. EFFPI is an embedded domain specific language in SCALA 3 that offers a simple Actor-based API for concurrency. Our code generation technique, as well the EFFPI library itself, leverages the type system features introduced in SCALA 3, e.g. match types and dependent function types, to encode local types in EFFPI. We extend EFFPI to support crash detection and handling.

As a brief introduction to EFFPI, the concurrency library provides types for processes and channels. For processes, an output process type `Out[A, B]` describes a process that uses a channel of type `A` to send a value of type `B`, and an input process type `In[A, B, C]` describes a process that uses a channel of type `A` to receive a value of type `B`, and pass it to a continuation type `C`. Process types can be sequentially composed by the `>>>` operator. For channels, `Chan[X]` describes a channel that can be used communicate values of type `X`. More specifically, the usage of a channel can be reflected at the type level, using the types `InChan[X]/OutChan[X]` for input/output channels.

```

1 type I[C0 <: InChan[Trigger], C1 <: OutChan[Fatal],
2     C2 <: InChan[Read], C3 <: InChan[Report], C4 <: OutChan[Report]]
3 = InErr[C0, Trigger,
4     (X <: Read) =>
5     Out[C3,Read] >>> In[C4, Report, (Y <: Log) => Out[C5, Report]],
6     (Err <: Throwable) => Out[C2,Fatal]
7 ]

```

■ **Figure 3** EFFPI Type for T_I .

As a sneak peek of the code we generate, in Fig. 3, we show the generated EFFPI representation for the projected local type T_I from the Simpler Logging example. Readers may be surprised by the difference between T_I and the generated EFFPI type `I`. This is because the process types need their respective channel types, namely the type variables `C0`, `C1`, *etc.* bounded by `InChan[...]` and `OutChan[...]`. We explain the details of code generation in §6.2, and describe an interesting challenge posed by the channel generation procedure in §6.3.

For crash handling behaviour, we introduce a new type `InErr`, whose last argument specifies a continuation type to follow in case of a crash. Line 3 in Fig. 3 shows the crash handling behaviour: sending a message of type `Fatal`, which reflects the `crash` branch in the local type T_I . We give more details of the generated code in §6.2.

Code generated by TEATRINO is executable, protocol-conforming, and can be specialised by the programmer to integrate with existing code. We evaluate our toolchain on examples taken from both MPST and distributed programming literature in §7. Moreover, we extend each example with crash handling behaviour to define unreliable variants. We demonstrate that, with TEATRINO, code generation takes negligible time, and all potential crashes are accompanied with crash handlers.

3 Crash-Stop Asynchronous Multiparty Session Calculus

In this section, we formalise the syntax and operational semantics of our asynchronous multiparty session calculus with process failures and crash detection.

Syntax. Our asynchronous multiparty session calculus models processes that may crash arbitrarily. Our formalisation is based on [16] – but in addition, follows the *fail-stop* model in [6, §2.7], where processes may crash and never recover, and process failures can be detected by failure detectors [6, §2.6.2] [8] when attempting to receive messages.

We give the syntax of processes in Fig. 4. In our calculus, we assume that there are basic expressions e (e.g. `true`, `false`, $7 + 11$) that are assigned basic types B (e.g. `int`, `bool`). We write $e \downarrow v$ to denote an expression e evaluates to a value v (e.g. $(7 < 11) \downarrow \text{true}$, $(1 + 1) \downarrow 2$).

$P, Q ::=$ $\sum_{i \in I} \mathbf{p}^? \mathbf{m}_i(x_i).P_i$ $\mathbf{p}! \mathbf{m}(e).P \quad (\text{where } \mathbf{m} \neq \text{crash})$ $\text{if } e \text{ then } P \text{ else } Q$ X $\mu X.P$ $\mathbf{0}$ ζ	Processes <i>external choice</i> <i>output</i> <i>conditional</i> <i>variable</i> <i>recursion</i> <i>inaction</i> <i>crashed</i>	$\mathcal{M} ::=$ $\mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h$ $\mathcal{M} \mid \mathcal{M}$ $h ::=$ ϵ \emptyset $(\mathbf{p}, \mathbf{m}(v))$ $h \cdot h$	Sessions <i>participant</i> <i>parallel</i> Queues <i>empty</i> <i>unavailable</i> <i>message</i> <i>concatenation</i>
---	---	--	---

■ **Figure 4** Syntax of sessions, processes, and queues. Noticeable changes w.r.t. standard session calculus [16] are highlighted.

A process, ranged over by P, Q , is a communication agent within a session. An *output* process $\mathbf{p}! \mathbf{m}(e).P$ sends a message to another role \mathbf{p} in the session, where the message is labelled \mathbf{m} , and carries a payload expression e , then the process continues as P . An *external choice (input)* process $\sum_{i \in I} \mathbf{p}^? \mathbf{m}_i(x_i).P_i$ receives a message from another role \mathbf{p} in the session, among a finite set of indexes I , if the message is labelled \mathbf{m}_i , then the payload would be received as x_i , and process continues as P_i . Note that our calculus uses *crash* as a special message label denoting that a participant (role) has crashed. Such a label cannot be sent by any process, but a process can implement crash detection and handling by receiving it. Consequently, an output process *cannot* send a *crash* message (side condition $\mathbf{m} \neq \text{crash}$), whereas an input process may include a *crash handling branch* of the form $\text{crash}.P'$ meaning that P' is executed when the sending role has crashed. A *conditional* process $\text{if } e \text{ then } P \text{ else } Q$ continues as either P or Q depending on the evaluation of e . We allow *recursion* at the process level using $\mu X.P$ and X , and we require process recursion variables to be guarded by an input or an output action; we consider a recursion process structurally congruent to its unfolding $\mu X.P \equiv P\{\mu X.P/X\}$. Finally, we write $\mathbf{0}$ for an *inactive* process, representing a successful termination; and ζ for a *crashed* process, representing a termination due to failure.

An *incoming queue*², ranged over by h, h' , is a sequence of messages tagged with their origin. We write ϵ for an *empty* queue; \emptyset for an *unavailable* queue; and $(\mathbf{p}, \mathbf{m}(v))$ for a message sent from \mathbf{p} , labelled \mathbf{m} , and containing a payload value v . We write $h_1 \cdot h_2$ to denote the concatenation of two queues h_1 and h_2 . When describing incoming queues, we consider two messages from different origins as swappable: $h_1 \cdot (\mathbf{q}_1, \mathbf{m}_1(v_1)) \cdot (\mathbf{q}_2, \mathbf{m}_2(v_2)) \cdot h_2 \equiv h_1 \cdot (\mathbf{q}_2, \mathbf{m}_2(v_2)) \cdot (\mathbf{q}_1, \mathbf{m}_1(v_1)) \cdot h_2$ whenever $\mathbf{q}_1 \neq \mathbf{q}_2$. Moreover, we consider concatenation (\cdot) as associative, and the empty queue ϵ as the identity element for concatenation.

A session, ranged over by $\mathcal{M}, \mathcal{M}'$, consists of processes and their respective incoming queue, indexed by their roles. A single entry for a role \mathbf{p} is denoted $\mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h$, where P is the process for \mathbf{p} and h is the incoming queue. Entries are composed together in parallel as $\mathcal{M} \mid \mathcal{M}'$, where the roles in \mathcal{M} and \mathcal{M}' are disjoint. We consider parallel composition as commutative and associative, with $\mathbf{p} \triangleleft \mathbf{0} \mid \mathbf{p} \triangleleft \epsilon$ as a neutral element of the operator. We write $\prod_{i \in I} (\mathbf{p}_i \triangleleft P_i \mid \mathbf{p}_i \triangleleft h_i)$ for the parallel composition of multiple entries in a set.

Operational Semantics. Operational Semantics of our session calculus is given in Def. 1, using a standard *structural congruence* \equiv defined in [16]. Our semantics parameterises on a (possibly empty) set of *reliable* roles \mathcal{R} , which are assumed to *never crash*.

² In [16], the queues are outgoing instead of incoming. We use incoming queues to model our crashing semantics more easily.

[R- \downarrow]	$\mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathcal{M} \rightarrow_{\mathcal{R}} \mathbf{p} \triangleleft \downarrow \mid \mathbf{p} \triangleleft \emptyset \mid \mathcal{M}$	$(P \neq \mathbf{0}, \mathbf{p} \notin \mathcal{R})$
[R-SEND]	$\mathbf{p} \triangleleft \mathbf{q}!m(e).P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft h_{\mathbf{q}} \mid \mathcal{M}$ $\rightarrow \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft h_{\mathbf{q}} \cdot (\mathbf{p}, m(v)) \mid \mathcal{M}$	$(e \downarrow v, h_{\mathbf{q}} \neq \emptyset)$
[R-SEND- \downarrow]	$\mathbf{p} \triangleleft \mathbf{q}!m(e).P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M}$	
[R-RCV]	$\mathbf{p} \triangleleft \sum_{i \in I} \mathbf{q}^?m_i(x_i).P_i \mid \mathbf{p} \triangleleft (\mathbf{q}, m_k(v)) \cdot h_{\mathbf{p}} \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft P_k\{v/x_k\} \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathcal{M}$	$(k \in I)$
[R-RCV- \emptyset]	$\mathbf{p} \triangleleft \sum_{i \in I} \mathbf{q}^?m_i(x_i).P_i \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M}$ $\rightarrow \mathbf{p} \triangleleft P_k \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M}$	$(k \in I, m_k = \text{crash}, \nexists m, v : (\mathbf{q}, m(v)) \in h_{\mathbf{p}})$
[R-COND-T]	$\mathbf{p} \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathbf{p} \triangleleft h \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h \mid \mathcal{M}$	$(e \downarrow \text{true})$
[R-COND-F]	$\mathbf{p} \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathbf{p} \triangleleft h \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft Q \mid \mathbf{p} \triangleleft h \mid \mathcal{M}$	$(e \downarrow \text{false})$
[R-STRUCT]	$\mathcal{M}_1 \equiv \mathcal{M}'_1 \text{ and } \mathcal{M}'_1 \rightarrow \mathcal{M}'_2 \text{ and } \mathcal{M}'_2 \equiv \mathcal{M}_2 \implies \mathcal{M}_1 \rightarrow \mathcal{M}_2$	

■ **Figure 5** Reduction relation on sessions with crash-stop failures.

► **Definition 1** (Session Reductions). *Session reduction $\rightarrow_{\mathcal{R}}$ is inductively defined by the rules in Fig. 5, parameterised by a fixed set \mathcal{R} of reliable roles. We write \rightarrow when \mathcal{R} is insignificant. We write $\rightarrow_{\mathcal{R}}^*$ (resp. \rightarrow^*) for the reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$ (resp. \rightarrow).*

Our operational semantics retains the basic rules in [16], but also includes (highlighted) rules for crash-stop failures and crash handling, adapted from [3]. Rules [R-SEND] and [R-RCV] model ordinary message delivery and reception: an output process located at \mathbf{p} sending to \mathbf{q} appends a message to the incoming queue of \mathbf{q} ; and an input process located at \mathbf{p} receiving from \mathbf{q} consumes the first message from the incoming queue. Rules [R-COND-T] and [R-COND-F] model conditionals; and rule [R-STRUCT] permits reductions up to structural congruence.

With regard to crashes and related behaviour, rule [R- \downarrow] models process crashes: an active ($P \neq \mathbf{0}$) process located at an unreliable role ($\mathbf{p} \notin \mathcal{R}$) may reduce to a crashed process $\mathbf{p} \triangleleft \downarrow$, with its incoming queue becoming unavailable $\mathbf{p} \triangleleft \emptyset$. Rule [R-SEND- \downarrow] models a message delivery to a crashed role (and thus an unavailable queue), and the message becomes lost and would not be added to the queue. Rule [R-RCV- \emptyset] models crash detection, which activates as a “last resort”: an input process at \mathbf{p} receiving from \mathbf{q} would first attempt find a message from \mathbf{q} in the incoming queue, which engages the usual rule [R-RCV]; if none exists and \mathbf{q} has crashed ($\mathbf{q} \triangleleft \downarrow$), then the crash handling branch in the input process at \mathbf{p} can activate. We draw attention to the interesting fact that [R-RCV] may engage even if \mathbf{q} has crashed, in cases where a message from \mathbf{q} in the incoming queue may be consumed. We now illustrate our operational semantics of sessions with an example.

► **Example 2.** Consider the session $\mathcal{M} = \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft \epsilon \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft \epsilon$, where $P = \mathbf{q}!m(\text{“abc”}). \sum \left\{ \mathbf{q}^?m'(x).\mathbf{0} \right\}$ and $Q = \sum \left\{ \mathbf{p}^?m(x).\mathbf{p}!m'(42).\mathbf{0} \right\}$. In this session, the process Q for \mathbf{q} receives a message sent from \mathbf{p} to \mathbf{q} ; the process P for \mathbf{p} sends a message from \mathbf{p} to \mathbf{q} , and then receives a message sent from \mathbf{q} to \mathbf{p} . Let each role be unreliable, i.e. $\mathcal{R} = \emptyset$, and P crash before sending. We have $\mathcal{M} \rightarrow_{\emptyset} \mathbf{p} \triangleleft \downarrow \mid \mathbf{p} \triangleleft \emptyset \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft \epsilon \rightarrow \mathbf{p} \triangleleft \downarrow \mid \mathbf{p} \triangleleft \emptyset \mid \mathbf{q} \triangleleft \mathbf{0} \mid \mathbf{q} \triangleleft \epsilon$. We observe that when the output process P located at an unreliable role \mathbf{p} crashes (by [R- \downarrow]), the resulting entry for \mathbf{p} is a crashed process ($\mathbf{p} \triangleleft \downarrow$) with an unavailable queue ($\mathbf{p} \triangleleft \emptyset$). Subsequently, the input process Q located at \mathbf{q} can detect and handle the crash by [R-RCV- \emptyset] via its crash handling branch.

B	$::=$	<code>int</code> <code>bool</code> <code>real</code> <code>unit</code> ...	Basic types
G	$::=$	<code>p→q†: {m_i(B_i).G_i}_{i∈I}</code>	Transmission
		<code>p†↔q:j {m_i(B_i).G_i}_{i∈I} (j ∈ I)</code>	Transmission en route
		<code>μt.G</code> <code>t</code> <code>end</code>	Recursion, Type variable, Termination
$†$	$::=$	<code>·</code> <code>⚡</code>	Crash annotation
S, T	$::=$	<code>p&{m_i(B_i).T_i}_{i∈I} <code>p⊕{m_i(B_i).T_i}_{i∈I}</code></code>	External choice, Internal choice
		<code>μt.T</code> <code>t</code> <code>end</code> <code>stop</code>	Recursion, Type variable, Termination, Crash

■ **Figure 6** Syntax of global types and local types. Runtime types are shaded.

4 Asynchronous Multiparty Session Types with Crash-Stop Semantics

In this section, we present our asynchronous multiparty session types with crash-stop semantics. We give an overview of global and local types with crashes in §4.1, including syntax, projection, subtyping, *etc.*; our key additions to the classic theory are *crash handling branches* in both global and local types, and a special local type `stop` to denote crashed processes. We give a Labelled Transition System (LTS) semantics to both global types (§4.2) and configurations (i.e. a collection of local types and point-to-point communication queues, §4.3). We discuss alternative design options of modelling crash-stop failures in §4.4. We relate the two semantics in §4.5, and show that a configuration obtained via projection is safe, deadlock-free, and live in §4.6.

4.1 Global and Local Types with Crash-Stop Failures

The top-down methodology begins with *global types* to provide an overview of the communication between a number of *roles* (p, q, s, t, \dots), belonging to a (fixed) set \mathcal{R} . At the other end, we use *local types* to describe how a *single* role communicates with other roles from a local perspective, and they are obtained via *projection* from a global type. We give the syntax of both global and local types in Fig. 6, which are similar to syntax used in [3, 37].

Global Types. Global Types are ranged over G, G', G_i, \dots , and describe the behaviour for all roles from a bird's eye view. The syntax shown in `shade` are *runtime* syntax, which are not used for describing a system at design-time, but for describing the state of a system during execution. The labels m are taken from a fixed set of all labels \mathcal{M} , and basic types B (types for payloads) from a fixed set of all basic types \mathcal{B} .

We explain each construct in the syntax of global types: a transmission, denoted $p \rightarrow q^\dagger: \{m_i(B_i).G_i\}_{i \in I}$, represents a message from role p to role q (with possible crash annotations), with labels m_i , payload types B_i , and continuations G_i , where i is taken from an index set I . We require that the index set be non-empty ($I \neq \emptyset$), labels m_i be pair-wise distinct, and self receptions be excluded (i.e. $p \neq q$), as standard in session type works. Additionally, we require that the special `crash` label (explained later) not be the only label in a transmission, i.e. $\{m_i \mid i \in I\} \neq \{\text{crash}\}$. A transmission en route $p^\dagger \leftrightarrow q:j \{m_i(B_i).G_i\}_{i \in I}$ is a runtime construct representing a message m_j (index j) sent by p , and yet to be received by q . Recursive types are represented via $\mu t.G$ and t , where contractive requirements apply [34, §21.8]. The type `end` describes a terminated type (omitted where unambiguous).

To model crashes and crash handling, we use crash annotations `⚡` and crash handling branches: a *crash annotation* `⚡`, a new addition in this work, marks a *crashed* role (only used in the *runtime syntax*), and we omit annotations for live roles, i.e. p is a live role, $p^{\text{⚡}}$ is

a crashed role, and p^\dagger represents a possibly crashed role, namely either p or p^\dagger . We use a special label `crash` for handling crashes: this continuation denotes the protocol to follow when the sender of a message is detected to have crashed by the receiver. The special label acts as a “pseudo-message”: when a sender role crashes, the receiver can select the pseudo-message to enter crash handling. We write $\text{roles}(G)$ (resp. $\text{roles}^\dagger(G)$) for the set of *active* (resp. *crashed*) roles in a global type G , *excluding* (resp. *consisting only of*) those with a crash annotation \dagger .

Local Types. Local Types are ranged over S, T, U, \dots , and describe the behaviour of a single role. An internal choice (selection) (or an external choice (branching)), denoted $p \oplus \{m_i(B_i).T_i\}_{i \in I}$ (or $p \& \{m_i(B_i).T_i\}_{i \in I}$), indicates that the *current* role is to *send* to (or *receive* from) the role p . Similarly to global types, we require pairwise-distinct, non-empty labels. Moreover, we require that the `crash` label not appear in *internal* choices, reflecting that a `crash` pseudo-message can never be sent; and that singleton `crash` labels not permitted in external choices. The type `end` indicates a *successful* termination (omitted where unambiguous), and recursive types follow a similar fashion to global types. We use a new *runtime* type `stop` to denote crashes.

Subtyping. Subtyping relation \leq on local types will be used in §4.5 to relate global and local type semantics. Our subtyping relation is mostly standard [37, Def. 2.5], except for an extra rule for `stop` and additional requirements to support crash handling branch in external choices.

Projection. Projection gives the local type of a participating role in a global type, defined as a *partial* function that takes a global type G and a role p , and returns a local type, given by Def. 3.

► **Definition 3** (Global Type Projection). *The projection of a global type G onto a role p , with respect to a set of reliable roles \mathcal{R} , written $G \upharpoonright_{\mathcal{R}} p$, is:*

$$\begin{aligned}
 (q \rightarrow r^\dagger : \{m_i(B_i).G_i\}_{i \in I}) \upharpoonright_{\mathcal{R}} p &= \begin{cases} r \oplus \{m_i(B_i).(G_i \upharpoonright_{\mathcal{R}} p)\}_{i \in \{j \in I \mid m_j \neq \text{crash}\}} & \text{if } p = q \\ q \& \{m_i(B_i).(G_i \upharpoonright_{\mathcal{R}} p)\}_{i \in I} & \text{if } p = r, \text{ and } q \notin \mathcal{R} \text{ implies } \\ & \exists k \in I : m_k = \text{crash} \\ \prod_{i \in I} G_i \upharpoonright_{\mathcal{R}} p & \text{if } p \neq q, \text{ and } p \neq r \end{cases} \\
 (q^\dagger \rightsquigarrow r : \{m_i(B_i).G_i\}_{i \in I}) \upharpoonright_{\mathcal{R}} p &= \begin{cases} G_j \upharpoonright_{\mathcal{R}} p & \text{if } p = q \\ q \& \{m_i(B_i).(G_i \upharpoonright_{\mathcal{R}} p)\}_{i \in I} & \text{if } p = r, \text{ and } q \notin \mathcal{R} \text{ implies } \\ & \exists k \in I : m_k = \text{crash} \\ \prod_{i \in I} G_i \upharpoonright_{\mathcal{R}} p & \text{if } p \neq q, \text{ and } p \neq r \end{cases} \\
 (\mu t. G) \upharpoonright_{\mathcal{R}} p &= \begin{cases} \mu t.(G \upharpoonright_{\mathcal{R}} p) & \text{if } p \in G \text{ or } \text{fv}(\mu t. G) \neq \emptyset \\ \text{end} & \text{otherwise} \end{cases} \quad \begin{matrix} t \upharpoonright_{\mathcal{R}} p = t \\ \text{end} \upharpoonright_{\mathcal{R}} p = \text{end} \end{matrix}
 \end{aligned}$$

where \prod is the merge operator for local types (full merging):

$$\begin{aligned}
 & p \& \{m_i(B_i).S'_i\}_{i \in I} \prod p \& \{m_j(B_j).T'_j\}_{j \in J} \\
 = & p \& \{m_k(B_k).(S'_k \prod T'_k)\}_{k \in I \cap J} \& p \& \{m_i(B_i).S'_i\}_{i \in I \setminus J} \& p \& \{m_j(B_j).T'_j\}_{j \in J \setminus I} \\
 & p \oplus \{m_i(B_i).S'_i\}_{i \in I} \prod p \oplus \{m_i(B_i).T'_i\}_{i \in I} = p \oplus \{m_i(B_i).(S'_i \prod T'_i)\}_{i \in I} \\
 & \mu t. S \prod \mu t. T = \mu t.(S \prod T) \quad t \prod t = t \quad \text{end} \prod \text{end} = \text{end}
 \end{aligned}$$

We parameterise our theory on a (fixed) set of *reliable* roles \mathcal{R} , i.e. roles assumed to *never crash*: if $\mathcal{R} = \emptyset$, every role is unreliable and susceptible to crash; if $\text{roles}(G) \subseteq \mathcal{R}$, every role in G is reliable, and we simulate the results from the original MPST theory without crashes. We base our definition of projection on [37], but include more (highlighted) cases to account for reliable roles, `crash` branches, and runtime global types.

When projecting a transmission from q to r , we remove the **crash** label from the internal choice at q , reflecting our model that a **crash** pseudo-message cannot be sent. Dually, we require a **crash** label to be present in the external choice at r – unless the sender role q is assumed to be reliable. Our definition of projection enforces that transmissions, whenever an unreliable role is the sender ($q \notin \mathcal{R}$), *must include* a crash handling branch ($\exists k \in I : m_k = \text{crash}$). This requirement ensures that the receiving role r can *always* handle crashes whenever it happens, so that processes are not stuck when crashes occur. We explain how these requirements help us achieve various properties by projection in §4.6. The rest of the rules are taken from the literature [37,40], without much modification.

4.2 Crash-Stop Semantics of Global Types

We now give a Labelled Transition System (LTS) semantics to global types, with crash-stop semantics. To this end, we first introduce some auxiliary definitions. We define the transition labels in Def. 4, which are also used in the LTS semantics of configurations (later in §4.3).

► **Definition 4** (Transition Labels). *Let α be a transition label of the form:*

$$\alpha ::= \begin{array}{l|l} p\&q:m(B) & (p \text{ receives } m(B) \text{ from } q) \\ \hline p\downarrow & (p \text{ crashes}) \end{array} \quad \left| \quad \begin{array}{l|l} p\oplus q:m(B) & (p \text{ sends } m(B) \text{ to } q) \\ \hline p\odot q & (p \text{ detects the crash of } q) \end{array}$$

The subject of a transition label, written $\text{subj}(\alpha)$, is defined as:

$$\text{subj}(p\&q:m(B)) = \text{subj}(p\oplus q:m(B)) = \text{subj}(p\downarrow) = \text{subj}(p\odot q) = p.$$

The labels $p\oplus q:m(B)$ and $p\&q:m(B)$ describe sending and receiving actions respectively. The crash of p is denoted by the label $p\downarrow$, and the detection of a crash by label $p\odot q$: we model crash detection at *reception*, the label contains a *detecting* role p and a *crashed* role q .

We define an operator to *remove* a role from a global type in Def. 5: the intuition is to remove any interaction of a crashed role from the given global type. When a role has crashed, we attach a *crashed annotation*, and remove infeasible actions, e.g. when the sender and receiver of a transmission have both crashed. The removal operator is a partial function that takes a global type G and a live role r ($r \in \text{roles}(G)$) and gives a global type $G\downarrow r$.

► **Definition 5** (Role Removal). *The removal of a live role p in a global type G , written $G\downarrow p$, is defined as follows:*

$$\begin{aligned} (p \rightarrow q : \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} p\downarrow \rightsquigarrow q:j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p = r \text{ and } \exists j \in I : m_j = \text{crash} \\ p \rightarrow q\downarrow : \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } q = r \\ p \rightarrow q : \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (p \rightsquigarrow q : j \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} p\downarrow \rightsquigarrow q:j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p = r \\ G_j\downarrow r & \text{if } q = r \\ p \rightsquigarrow q : j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (p \rightarrow q\downarrow : \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} G_j\downarrow r & \text{if } p = r \text{ and } \exists j \in I : m_j = \text{crash} \\ p \rightarrow q\downarrow : \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (p\downarrow \rightsquigarrow q : j \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} G_j\downarrow r & \text{if } q = r \\ p\downarrow \rightsquigarrow q : j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (\mu t.G)\downarrow r &= \begin{cases} \mu t.(G\downarrow r) & \text{if } \text{fv}(\mu t.G) \neq \emptyset \text{ or } \text{roles}(G\downarrow r) \neq \emptyset \\ \text{end} & \text{otherwise} \end{cases} \\ t\downarrow r &= t & \text{end}\downarrow r &= \text{end} \end{aligned}$$

For simple cases, the removal of a role $G\downarrow r$ attaches crash annotations \downarrow on all occurrences of the removed role r throughout global type G inductively.

$$\begin{array}{c}
 \frac{\mathbf{p} \notin \mathcal{R} \quad \mathbf{p} \in \text{roles}(G) \quad G \neq \mu\mathbf{t}.G'}{\langle \mathcal{C}; G \rangle \xrightarrow{\mathbf{p}^\ddagger} \langle \mathcal{C} \cup \{\mathbf{p}\}; G \downarrow \mathbf{p} \rangle} \text{ [GR-}\ddagger\text{]} \quad \frac{\langle \mathcal{C}; G\{\mu\mathbf{t}.G/\mathbf{t}\} \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G' \rangle}{\langle \mathcal{C}; \mu\mathbf{t}.G \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G' \rangle} \text{ [GR-}\mu\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j \neq \text{crash}}{\langle \mathcal{C}; \mathbf{p} \rightarrow \mathbf{q}; \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{p} \oplus \mathbf{q}_j(B_j)} \langle \mathcal{C}; \mathbf{p} \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle} \text{ [GR-}\oplus\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j \neq \text{crash}}{\langle \mathcal{C}; \mathbf{p}^\dagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{q} \& \mathbf{m}_j(B_j)} \langle \mathcal{C}; G'_j \rangle} \text{ [GR-}\&\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j = \text{crash}}{\langle \mathcal{C}; \mathbf{p}^\ddagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{q} \circ \mathbf{p}} \langle \mathcal{C}; G'_j \rangle} \text{ [GR-}\circ\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j \neq \text{crash}}{\langle \mathcal{C}; \mathbf{p} \rightarrow \mathbf{q}^\ddagger; \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{p} \oplus \mathbf{q}_j(B_j)} \langle \mathcal{C}; G'_j \rangle} \text{ [GR-}\ddagger \oplus\text{]} \\
 \frac{\forall i \in I : \langle \mathcal{C}; G'_i \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G''_i \rangle \quad \text{subj}(\alpha) \notin \{\mathbf{p}, \mathbf{q}\}}{\langle \mathcal{C}; \mathbf{p} \rightarrow \mathbf{q}^\dagger; \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; \mathbf{p} \rightarrow \mathbf{q}^\dagger; \{ \mathbf{m}_i(B_i).G''_i \}_{i \in I} \rangle} \text{ [GR-Ctx-I]} \\
 \frac{\forall i \in I : \langle \mathcal{C}; G'_i \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G''_i \rangle \quad \text{subj}(\alpha) \neq \mathbf{q}}{\langle \mathcal{C}; \mathbf{p}^\dagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; \mathbf{p}^\dagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G''_i \}_{i \in I} \rangle} \text{ [GR-Ctx-II]}
 \end{array}$$

■ **Figure 7** Global Type Reduction Rules.

We draw attention to some interesting cases: when we remove the sender role \mathbf{p} from a transmission prefix $\mathbf{p} \rightarrow \mathbf{q}$, the result is a “pseudo-transmission” en route prefix $\mathbf{p}^\ddagger \rightsquigarrow \mathbf{q}; j$ where $\mathbf{m}_j = \text{crash}$. This enables the receiver \mathbf{q} to “receive” the special crash after the crash of \mathbf{p} , hence triggering the crash handling branch. Recall that our definition of projection requires that a crash handling branch be present whenever a crash may occur ($\mathbf{q} \notin \mathcal{R}$).

When we remove the sender role \mathbf{p} from a transmission en route prefix $\mathbf{p} \rightsquigarrow \mathbf{q}; j$, the result *retains* the index j that was selected by \mathbf{p} , instead of the index associated with crash handling. This is crucial to our crash modelling: when a role crashes, the messages that the role *has sent* to other roles are still available. We discuss alternative models later in § 4.4.

In other cases, where removing the role \mathbf{r} would render a transmission (regardless of being en route or not) meaningless, e.g. both sender and receiver have crashed, we simply remove the prefix entirely.

We now give an LTS semantics to a global type G , by defining the semantics with a tuple $\langle \mathcal{C}; G \rangle$, where \mathcal{C} is a set of *crashed* roles. The transition system is parameterised by reliability assumptions, in the form of a fixed set of reliable roles \mathcal{R} . When unambiguous, we write G as an abbreviation of $\langle \emptyset; G \rangle$. We define the reduction rules of global types in Def. 6.

► **Definition 6** (Global Type Reductions). *The global type (annotated with a set of crashed roles \mathcal{C}) transition relation $\xrightarrow{\alpha} \mathcal{R}$ is inductively defined by the rules in Fig. 7, parameterised by a fixed set \mathcal{R} of reliable roles. We write $\langle \mathcal{C}; G \rangle \rightarrow \mathcal{R} \langle \mathcal{C}'; G' \rangle$ if there exists α such that $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha} \mathcal{R} \langle \mathcal{C}'; G' \rangle$; we write $\langle \mathcal{C}; G \rangle \rightarrow \mathcal{R}$ if there exists \mathcal{C}', G' , and α such that $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha} \mathcal{R} \langle \mathcal{C}'; G' \rangle$, and $\rightarrow \mathcal{R}^*$ for the transitive and reflexive closure of $\rightarrow \mathcal{R}$.*

Rules [GR- \oplus] and [GR- $\&$] model sending and receiving messages respectively, as are standard in existing works [13]. We add an (highlighted) extra condition that the message exchanged not be a pseudo-message carrying the crash label. [GR- μ] is a standard rule handling recursion.

We introduce (highlighted) rules to account for crash and consequential behaviour. Rule [GR- \ddagger] models crashes, where a live ($\mathbf{p} \in \text{roles}(G)$), but unreliable ($\mathbf{p} \notin \mathcal{R}$) role \mathbf{p} may crash. The crashed role \mathbf{p} is added into the set of crashed roles ($\mathcal{C} \cup \{\mathbf{p}\}$), and removed from the global type, resulting in a global type $G \downarrow \mathbf{p}$. Rule [GR- \circ] is for *crash detection*, where a

live role q may detect that p has crashed at reception, and then continues with the crash handling continuation labelled `crash`. This rule only applies when the message en route is a pseudo-message, since otherwise a message rests in the queue of the receiver and can be received despite the crash of the sender (cf. $[\text{GR-}\&]$). Rule $[\text{GR-}\&^{\dagger}]$ models the orphaning of a message sent from a live role p to a crashed role q . Similar to the requirement in $[\text{GR-}\oplus]$, we add the side condition that the message sent is not a pseudo-message.

Finally, rules $[\text{GR-CTX-I}]$ and $[\text{GR-CTX-II}]$ allow non-interfering reductions of (intermediate) global types under prefix, provided that all of the continuations can be reduced by that label.

► **Remark 7 (Necessity of \mathcal{C} in Semantics).** While we can obtain the set of crashed roles in any global type G via $\text{roles}^{\dagger}(G)$, we need a separate \mathcal{C} for bookkeeping purposes. To illustrate, let $G = p \rightarrow q: \{m.\text{end}, \text{crash.end}\}$; we can have the following reductions:

$$\langle \emptyset; G \rangle \xrightarrow{q^{\dagger}} \langle \{q\}; p \rightarrow q^{\dagger}: \{m.\text{end}, \text{crash.end}\} \rangle \xrightarrow{p \oplus q: m} \langle \{q\}; \text{end} \rangle$$

While we can deduce q is a crashed role in the interim global type, the same information cannot be recovered from the final global type `end`.

4.3 Crash-Stop Semantics of Configurations

After giving semantics to global types, we now give an LTS semantics to *configurations*, i.e. a collection of local types and communication queues across roles. We first give a definition of configurations in Def. 8, followed by their reduction rules in Def. 9.

► **Definition 8 (Configurations).** A configuration is a tuple $\Gamma; \Delta$, where Γ is a typing context, denoting a partial mapping from roles to local types, defined as: $\Gamma ::= \emptyset \mid \Gamma, p \triangleright T$. We write $\Gamma[p \mapsto T]$ for updates: $\Gamma[p \mapsto T](p) = T$ and $\Gamma[p \mapsto T](q) = \Gamma(q)$ (where $p \neq q$).

A queue, denoted τ , is either a (possibly empty) sequence of messages $M_1 \cdot M_2 \cdots M_n$, or unavailable \circ . We write ϵ for an empty queue, and $M \cdot \tau'$ for a non-empty queue with message M at the beginning. A queue message M is of form $m(B)$, denoting a message with label m and payload B . We sometimes omit B when the payload is not of specific interest.

We write Δ to denote a queue environment, a collection of peer-to-peer queues. A queue from p to q at Δ is denoted $\Delta(p, q)$. We define updates $\Delta[p, q \mapsto \tau]$ similarly. We write Δ_{\emptyset} for an empty queue environment, where $\Delta_{\emptyset}(p, q) = \epsilon$ for any p and q in the domain.

We write $\tau' \cdot M$ to append a message M at the end of a queue τ' : the message is appended to the sequence when τ' is available, or discarded when τ' is unavailable (i.e. $\circ \cdot M = \circ$). Additionally, we write $\Delta[\cdot, q \mapsto \circ]$ for making all the queues to q unavailable: i.e. $\Delta[p_1, q \mapsto \circ][p_2, q \mapsto \circ] \cdots [p_n, q \mapsto \circ]$.

We give an LTS semantics of configurations in Def. 9. Similar to that of global types, we model the semantics of configurations in an asynchronous (a.k.a. message passing) fashion, using a queue environment to represent the communication queues among all roles.

► **Definition 9 (Configuration Semantics).** The configuration transition relation $\xrightarrow{\alpha}$ is defined in Fig. 8. We write $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$ iff $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$ for some Γ' and Δ' . We define two reductions \rightarrow and $\rightarrow_{\mathcal{R}}$ (where \mathcal{R} is a fixed set of reliable roles) as follows:

- We write $\Gamma; \Delta \rightarrow \Gamma'; \Delta'$ for $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$ with $\alpha \in \{p \& q: m(B), p \oplus q: m(B), p \circ q\}$. We write $\Gamma; \Delta \rightarrow$ iff $\Gamma; \Delta \rightarrow \Gamma'; \Delta'$ for some $\Gamma'; \Delta'$, and $\Gamma; \Delta \not\rightarrow$ for its negation, and \rightarrow^* for the reflexive and transitive closure of \rightarrow ;
- We write $\Gamma; \Delta \rightarrow_{\mathcal{R}} \Gamma'; \Delta'$ for $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$ with $\alpha \notin \{r^{\dagger} \mid r \in \mathcal{R}\}$. We write $\Gamma; \Delta \rightarrow_{\mathcal{R}}$ iff $\Gamma; \Delta \rightarrow_{\mathcal{R}} \Gamma'; \Delta'$ for some $\Gamma'; \Delta'$, and $\Gamma; \Delta \not\rightarrow_{\mathcal{R}}$ for its negation. We define $\rightarrow_{\mathcal{R}}^*$ as the reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$.

$$\begin{array}{c}
\frac{\Gamma(\mathbf{p}) = \mathbf{q} \oplus \{\mathbf{m}_i(B_i).T_i\}_{i \in I} \quad k \in I}{\Gamma; \Delta \xrightarrow{\mathbf{p} \oplus \mathbf{q}; \mathbf{m}_k(B_k)} \Gamma[\mathbf{p} \mapsto T_k]; \Delta[\mathbf{p}, \mathbf{q} \mapsto \Delta(\mathbf{p}, \mathbf{q}) \cdot \mathbf{m}_k(B_k)]} \text{[}\Gamma\text{-}\oplus\text{]} \\
\frac{\Gamma(\mathbf{p}) = \mathbf{q} \& \{\mathbf{m}_i(B_i).T_i\}_{i \in I} \quad k \in I \quad \mathbf{m}' = \mathbf{m}_k \quad B' = B_k \quad \Delta(\mathbf{q}, \mathbf{p}) = \mathbf{m}'(B') \cdot \tau' \neq \emptyset}{\Gamma; \Delta \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}_k(B_k)} \Gamma[\mathbf{p} \mapsto T_k]; \Delta[\mathbf{q}, \mathbf{p} \mapsto \tau']} \text{[}\Gamma\text{-}\&\text{]} \\
\frac{\Gamma(\mathbf{p}) = \mu t. T \quad \Gamma[\mathbf{p} \mapsto T\{\mu t. T/t\}]; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'}{\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'} \text{[}\Gamma\text{-}\mu\text{]} \quad \frac{\Gamma(\mathbf{p}) \neq \text{end} \quad \Gamma(\mathbf{p}) \neq \text{stop}}{\Gamma; \Delta \xrightarrow{\mathbf{p} \ddagger} \Gamma[\mathbf{p} \mapsto \text{stop}]; \Delta[\cdot, \mathbf{p} \mapsto \emptyset]} \text{[}\Gamma\text{-}\ddagger\text{]} \\
\frac{\Gamma(\mathbf{q}) = \mathbf{p} \& \{\mathbf{m}_i(B_i).T_i\}_{i \in I} \quad \Gamma(\mathbf{p}) = \text{stop} \quad k \in I \quad \mathbf{m}_k = \text{crash} \quad \Delta(\mathbf{p}, \mathbf{q}) = \epsilon}{\Gamma; \Delta \xrightarrow{\mathbf{q} \odot \mathbf{p}} \Gamma[\mathbf{q} \mapsto T_k]; \Delta} \text{[}\Gamma\text{-}\odot\text{]}
\end{array}$$

■ **Figure 8** Configuration Semantics.

We first explain the standard rules: rule $[\Gamma\text{-}\oplus]$ (resp. $[\Gamma\text{-}\&]$) says that a role can perform an output (resp. input) transition by appending (resp. consuming) a message at the corresponding queue. Recall that whenever a queue is unavailable, the resulting queue remains unavailable after appending ($\emptyset \cdot M = \emptyset$). Therefore, the rule $[\Gamma\text{-}\oplus]$ covers delivery to both crashed and live roles, whereas two separate rules are used in modelling global type semantics ($[\text{GR}\text{-}\oplus]$ and $[\text{GR}\text{-}\ddagger\text{m}]$). We also include a standard rule $[\Gamma\text{-}\mu]$ for recursive types.

The key innovations are the (highlighted) rules modelling crashes and crash detection: by rule $[\Gamma\text{-}\ddagger]$, a role \mathbf{p} may crash and become **stop** at any time (unless it is already **ended** or **stopped**). All of \mathbf{p} 's receiving queues become unavailable \emptyset , so that future messages to \mathbf{p} would be discarded. Rule $[\Gamma\text{-}\odot]$ models crash detection and handling: if \mathbf{p} is crashed and stopped, another role \mathbf{q} attempting to receive from \mathbf{p} can then take its **crash** handling branch. However, this rule only applies when the corresponding queue is empty: it is still possible to receive messages sent before crashing via $[\Gamma\text{-}\&]$.

4.4 Alternative Modellings for Crash-Stop Failures

Before we dive into the relation between two semantics, let us have a short digression to discuss our modelling choices and alternatives. In this work, we mostly follow the assumptions laid out in [3], where a crash is detected at reception. However, they opt to use a synchronous (rendez-vous) semantics, whereas we give an asynchronous (message passing) semantics, which entails interesting scenarios that would not arise in a synchronous semantics.

Specifically, consider the case where a role \mathbf{p} sends a message to \mathbf{q} , and then \mathbf{p} crashes after sending, but before \mathbf{q} receives the message. The situation does not arise under a synchronous semantics, since sending and receiving actions are combined into a single transmission action.

Intuitively, there are two possibilities to handle this scenario. The questions are whether the message sent immediately before crashing is deliverable to \mathbf{q} , and consequentially, at what time does \mathbf{q} detect the crash of \mathbf{p} .

In our semantics (Figs. 7 and 8), we opt to answer the first question in positive: we argue that this model is more consistent with our “passive” crash detection design. For example, if a role \mathbf{p} never receives from another role \mathbf{q} , then \mathbf{p} does not need to react in the event of \mathbf{q} 's crash. Following a similar line of reasoning, if the message sent by \mathbf{p} arrives in the receiving queue of \mathbf{q} , then \mathbf{q} should be able to receive the message, without triggering a crash detection (although it may be triggered later). As a consequence, we require in $[\Gamma\text{-}\odot]$ that the queue $\Delta(\mathbf{p}, \mathbf{q})$ be empty, to reflect the idea that crash detection should be a “last resort”.

For an alternative model, we can opt to detect the crash after crash has occurred. This is possibly better modelled with using outgoing queues (cf. [12]), instead of incoming queues in the semantics presented. Practically, this may be the scenario that a TCP connection is closed (or reset) when a peer has crashed, and the content in the queue is lost. It is worth noting that this kind of alternative model will not affect our main theoretical results: the operational correspondence between global and local type semantics, and furthermore, global type properties guaranteed by projection.

4.5 Relating Global Type and Configuration Semantics

We have given LTS semantics for both global types (Def. 6) and configurations (Def. 9), we now relate these two semantics with the help of the projection operator \uparrow (Def. 3).

We associate configurations $\Gamma; \Delta$ with global types G (as annotated with a set of crashed roles \mathcal{C}) by projection, written $\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$. Naturally, there are two components of the association: (1) the local types in Γ need to correspond to the projections of the global type G and the set of crashed roles \mathcal{C} ; and (2) the queues in Δ corresponds to the transmissions en route in the global type G and also the set of crashed roles \mathcal{C} .

► **Definition 10** (Association of Global Types and Configurations). *A configuration $\Gamma; \Delta$ is associated with a (well-annotated w.r.t. \mathcal{R}) global type $\langle \mathcal{C}; G \rangle$, written $\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$, iff*

1. Γ can be split into disjoint (possibly empty) sub-contexts $\Gamma = \Gamma_G, \Gamma_{\mathcal{C}}, \Gamma_{\text{end}}$ where:
 - (A1) Γ_G contains projections of G : $\text{dom}(\Gamma_G) = \text{roles}(G)$, and $\forall \mathbf{p} \in \text{dom}(\Gamma_G) : \Gamma(\mathbf{p}) \leq G \uparrow_{\mathcal{R}} \mathbf{p}$;
 - (A2) $\Gamma_{\mathcal{C}}$ contains crashed roles: $\text{dom}(\Gamma_{\mathcal{C}}) = \mathcal{C}$, and $\forall \mathbf{p} \in \text{dom}(\Gamma_{\mathcal{C}}) : \Gamma(\mathbf{p}) = \text{stop}$;
 - (A3) Γ_{end} contains only end endpoints: $\forall \mathbf{p} \in \Gamma_{\text{end}} : \Gamma(\mathbf{p}) = \text{end}$.
2. (A4) Δ is associated with global type $\langle \mathcal{C}; G \rangle$, given as follows:
 - i. Receiving queues for a role is unavailable if and only if it has crashed: $\forall \mathbf{q} : \mathbf{q} \in \mathcal{C} \iff \Delta(\cdot, \mathbf{q}) = \emptyset$;
 - ii. If $G = \text{end}$ or $G = \mu \mathbf{t}. G'$, then queues between all roles are empty (except receiving queue for crashed roles): $\forall \mathbf{p}, \mathbf{q} : \mathbf{q} \notin \mathcal{C} \implies \Delta(\mathbf{p}, \mathbf{q}) = \epsilon$;
 - iii. If $G = \mathbf{p} \rightarrow \mathbf{q}^{\dagger} : \{m_i(B_i).G'_i\}_{i \in I}$, or $G = \mathbf{p}^{\dagger} \rightsquigarrow \mathbf{q} : j \{m_i(B_i).G'_i\}_{i \in I}$ with $m_j = \text{crash}$ (i.e. a pseudo-message is en route), then
 - (i) if \mathbf{q} is live, then the queue from \mathbf{p} to \mathbf{q} is empty: $\mathbf{q}^{\dagger} \neq \mathbf{q}^{\ddagger} \implies \Delta(\mathbf{p}, \mathbf{q}) = \epsilon$, and
 - (ii) $\forall i \in I : \Delta$ is associated with $\langle \mathcal{C}; G'_i \rangle$;
 - iv. If $G = \mathbf{p}^{\dagger} \rightsquigarrow \mathbf{q} : j \{m_i(B_i).G'_i\}_{i \in I}$ with $m_j \neq \text{crash}$, then
 - (i) the queue from \mathbf{p} to \mathbf{q} begins with the message $m_j(B_j) : \Delta(\mathbf{p}, \mathbf{q}) = m_j(B_j) \cdot \tau$;
 - (ii) $\forall i \in I : \text{removing the message from the head of the queue, } \Delta[\mathbf{p}, \mathbf{q} \mapsto \tau]$ is associated with $\langle \mathcal{C}; G'_i \rangle$.

We write $\Gamma \sqsubseteq_{\mathcal{R}} G$ as an abbreviation of $\Gamma; \Delta_{\emptyset} \sqsubseteq_{\mathcal{R}} \langle \emptyset; G \rangle$. We sometimes say Γ (resp. Δ) is associated with $\langle \mathcal{C}; G \rangle$ for stating Item 1 (resp. Item 2) is satisfied.

We demonstrate the relation between the two semantics via association, by showing two main theorems: all possible reductions of a configuration have a corresponding action in reductions of the associated global type (Thm. 11); and the reducibility of a global type is the same as its associated configuration (Thm. 12).

► **Theorem 11** (Completeness of Association). *Given associated global type G and configuration $\Gamma; \Delta : \Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$. If $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$, where $\alpha \neq \mathbf{p}^{\ddagger}$ for all $\mathbf{p} \in \mathcal{R}$, then there exists $\langle \mathcal{C}'; G' \rangle$ such that $\Gamma'; \Delta' \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$ and $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha}_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$.*

► **Theorem 12** (Soundness of Association). *Given associated global type G and configuration $\Gamma; \Delta: \Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$. If $\langle \mathcal{C}; G \rangle \rightarrow_{\mathcal{R}}$, then there exists $\Gamma'; \Delta'$, α and $\langle \mathcal{C}'; G' \rangle$, such that $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha}_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$, $\Gamma'; \Delta' \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$, and $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$.*

By Thms. 11 and 12, we obtain, as a corollary, that a global type G is in operational correspondence with the typing context $\Gamma = \{\mathbf{p} \triangleright G \upharpoonright_{\mathcal{R}} \mathbf{p}\}_{\mathbf{p} \in \text{roles}(G)}$, which contains the projections of all roles in G .

4.6 Properties Guaranteed by Projection

A key benefit of our top-down approach of multiparty protocol design is that desirable properties are guaranteed by the methodology. As a consequence, processes following the local types obtained from projections are correct *by construction*. In this subsection, we focus on three properties: *communication safety*, *deadlock-freedom*, and *liveness*, and show that the three properties are guaranteed from configurations associated with global types.

Communication Safety. We begin by defining communication safety for configurations (Def. 13). We focus on two safety requirements:

- (i) each role must be able to handle any message that may end up in their receiving queue (so that there are no label mismatches); and
- (ii) each receiver must be able to handle the potential crash of the sender, unless the sender is reliable.

► **Definition 13** (Configuration Safety). *Given a fixed set of reliable roles \mathcal{R} , we say that φ is an \mathcal{R} -safety property of configurations iff, whenever $\varphi(\Gamma; \Delta)$, we have:*

[S- $\oplus \&$] $\Gamma(\mathbf{q}) = \mathbf{p} \& \{m_i(B_i).S'_i\}_{i \in I}$ and $\Delta(\mathbf{p}, \mathbf{q}) \neq \emptyset$ and $\Delta(\mathbf{p}, \mathbf{q}) \neq \epsilon$ implies $\Gamma; \Delta \xrightarrow{\mathbf{q} \& \mathbf{p}: m'(B')}$;

[S- $\& \&$] $\Gamma(\mathbf{p}) = \text{stop}$ and $\Gamma(\mathbf{q}) = \mathbf{p} \& \{m_i(S_i).S'_i\}_{i \in I}$ and $\Delta(\mathbf{p}, \mathbf{q}) = \epsilon$ implies $\Gamma; \Delta \xrightarrow{\mathbf{q} \oplus \mathbf{p}}$;

[S- μ] $\Gamma(\mathbf{p}) = \mu t.S$ implies $\varphi(\Gamma[\mathbf{p} \mapsto S\{\mu t.S/t\}]; \Delta)$;

[S- \rightarrow_i] $\Gamma; \Delta \rightarrow_{\mathcal{R}} \Gamma'; \Delta'$ implies $\varphi(\Gamma'; \Delta')$.

We say $\Gamma; \Delta$ is \mathcal{R} -safe, if $\varphi(\Gamma; \Delta)$ holds for some \mathcal{R} -safety property φ .

We use a coinductive view of the safety property [35], where the predicate of \mathcal{R} -safe configurations is the largest \mathcal{R} -safety property, by taking the union of all safety properties φ . For a configuration $\Gamma; \Delta$ to be \mathcal{R} -safe, it has to satisfy all clauses defined in Def. 13.

By clause [S- $\oplus \&$], whenever a role \mathbf{q} receives from another role \mathbf{p} , and a message is present in the queue, the receiving action must be possible for some label m' . Clause [S- $\& \&$] states that if a role \mathbf{q} receives from a crashed role \mathbf{p} , and there is nothing in the queue, then \mathbf{q} must have a *crash* branch, and a crash detection action can be fired. (Note that [S- $\oplus \&$] applies when the queue is non-empty, despite the crash of sender \mathbf{p} .) Finally, clause [S- μ] extends the previous clauses by unfolding any recursive entries; and clause [S- \rightarrow_i] states that any configuration $\Gamma'; \Delta'$ which $\Gamma; \Delta$ transitions to must also be \mathcal{R} -safe. By using transition $\rightarrow_{\mathcal{R}}$, we ignore crash transitions $\mathbf{p} \not\rightarrow$ for any reliable role $\mathbf{p} \in \mathcal{R}$.

► **Example 14.** Recall the local types $T_{\mathbf{C}}$, $T_{\mathbf{L}}$, and $T_{\mathbf{I}}$ of the Simpler Logging example in § 2. The configuration $\Gamma; \Delta$, where $\Gamma = \mathbf{C} \triangleright T_{\mathbf{C}}, \mathbf{L} \triangleright T_{\mathbf{L}}, \mathbf{I} \triangleright T_{\mathbf{I}}$ and $\Delta = \Delta_{\emptyset}$, is $\{\mathbf{L}, \mathbf{I}\}$ -safe. This can be verified by checking its reductions. For example, in the case where \mathbf{C} crashes immediately, we have: $\Gamma; \Delta \xrightarrow{\mathbf{C} \not\rightarrow} \Gamma[\mathbf{C} \mapsto \text{stop}]; \Delta[\cdot, \mathbf{C} \mapsto \emptyset] \xrightarrow{*} \Gamma[\mathbf{C} \mapsto \text{stop}][\mathbf{L} \mapsto \text{end}][\mathbf{I} \mapsto \text{end}]; \Delta[\cdot, \mathbf{C} \mapsto \emptyset]$ and each reductum satisfies all clauses of Def. 13.

Deadlock-Freedom. The property of deadlock-freedom, sometimes also known as progress, describes whether a configuration can keep reducing unless it is a terminal configuration. We give its formal definition in Def. 15.

► **Definition 15** (Configuration Deadlock-Freedom). *Given a set of reliable roles \mathcal{R} , we say that a configuration $\Gamma; \Delta$ is \mathcal{R} -deadlock-free iff:*

1. $\Gamma; \Delta$ is \mathcal{R} -safe; and,
2. If $\Gamma; \Delta$ can reduce to a configuration $\Gamma'; \Delta'$ without further reductions: $\Gamma; \Delta \rightarrow_{\mathcal{R}}^* \Gamma'; \Delta' \not\rightarrow_{\mathcal{R}}$, then:
 - a. Γ' can be split into two disjoint contexts, one with only **end** entries, and one with only **stop** entries: $\Gamma' = \Gamma'_{\text{end}}; \Gamma'_{\text{z}}$, where $\text{dom}(\Gamma'_{\text{end}}) = \{\mathbf{p} \mid \Gamma'(\mathbf{p}) = \text{end}\}$ and $\text{dom}(\Gamma'_{\text{z}}) = \{\mathbf{p} \mid \Gamma'(\mathbf{p}) = \text{stop}\}$; and,
 - b. Δ' is empty for all pairs of roles, except for the receiving queues of crashed roles, which are unavailable: $\forall \mathbf{p}, \mathbf{q} : \Delta'(\cdot, \mathbf{q}) = \emptyset$ if $\Gamma'(\mathbf{q}) = \text{stop}$, and $\Delta'(\mathbf{p}, \mathbf{q}) = \epsilon$, otherwise.

It is worth noting that a (safe) configuration that reduces infinitely satisfies deadlock-freedom, as Item 2 in the premise does not hold. Otherwise, whenever a terminal configuration is reached, it must satisfy Item 2a that all local types in the typing context be terminated (either successfully **end**, or crashed **stop**), and Item 2b that all queues be empty (unless unavailable due to crash). As a consequence, a deadlock-free configuration $\Gamma; \Delta$ either does not stop reducing, or terminates in a stable configuration.

Liveness. The property of liveness describes that every pending output/external choice is eventually triggered by means of a message transmission or crash detection. Our liveness property is based on *fairness*, which guarantees that every enabled message transmission, including crash detection, is performed successfully. We give the definitions of non-crashing, fair, and live paths of configurations respectively in Def. 16, and use these paths to formalise the liveness for configurations in Def. 17.

► **Definition 16** (Non-crashing, Fair, Live Paths). *A non-crashing path is a possibly infinite sequence of configurations $(\Gamma_n; \Delta_n)_{n \in N}$, where $N = \{0, 1, 2, \dots\}$ is a set of consecutive natural numbers, and $\forall n \in N, \Gamma_n; \Delta_n \rightarrow \Gamma_{n+1}; \Delta_{n+1}$. We say that a non-crashing path $(\Gamma_n; \Delta_n)_{n \in N}$ is fair iff, $\forall n \in N$:*

- (F1) $\Gamma_n; \Delta_n \xrightarrow{\mathbf{p} \oplus \mathbf{q}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$ implies $\exists k, \mathbf{m}', B'$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \oplus \mathbf{q}; \mathbf{m}'(B')} \Gamma_{k+1}; \Delta_{k+1}$;
- (F2) $\Gamma_n; \Delta_n \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$ implies $\exists k$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$;
- (F3) $\Gamma_n; \Delta_n \xrightarrow{\mathbf{p} \odot \mathbf{q}} \Gamma_{k+1}; \Delta_{k+1}$ implies $\exists k$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \odot \mathbf{q}} \Gamma_{k+1}; \Delta_{k+1}$.

We say that a non-crashing path $(\Gamma_n; \Delta_n)_{n \in N}$ is live iff, $\forall n \in N$:

- (L1) $\Delta_n(\mathbf{p}, \mathbf{q}) = \mathbf{m}(B) \cdot \tau \neq \emptyset$ and $\mathbf{m} \neq \text{crash}$ implies $\exists k$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{q} \& \mathbf{p}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$;
- (L2) $\Gamma_n(\mathbf{p}) = \mathbf{q} \& \{ \mathbf{m}_i(B_i) \cdot T_i \}_{i \in I}$ implies $\exists k, \mathbf{m}', B'$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}'(B')} \Gamma_{k+1}; \Delta_{k+1}$ or $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \odot \mathbf{q}} \Gamma_{k+1}; \Delta_{k+1}$.

A non-crash path is a (possibly infinite) sequence of reductions of a configuration without crashes. A non-crash path is fair if along the path, every internal choice eventually sends a message (F1), every external choice eventually receives a message (F2), and every crash detection is eventually performed (F3). A non-crashing path is live if along the path, every non-crash message in the queue is eventually consumed (L1), and every hanging external choice eventually consumes a message or performs a crash detection (L2).

$$\begin{array}{c}
 \frac{}{\vdash \epsilon : \epsilon} \text{[T-}\epsilon\text{]} \quad \frac{}{\vdash \emptyset : \emptyset} \text{[T-}\emptyset\text{]} \quad \frac{\vdash h_1 : \delta_1 \quad \vdash h_2 : \delta_2}{\vdash h_1 \cdot h_2 : \delta_1 \cdot \delta_2} \text{[T-}\cdot\text{]} \\
 \frac{\vdash v : B \quad \delta(\mathbf{q}) = \mathbf{m}(B) \quad \forall \mathbf{r} \neq \mathbf{q} : \delta(\mathbf{r}) = \epsilon}{\vdash (\mathbf{q}, \mathbf{m}(v)) : \delta} \text{[T-MSG]} \\
 \\
 \frac{\frac{\frac{}{\Theta \vdash \zeta : \text{stop}} \text{[T-}\zeta\text{]} \quad \frac{}{\Theta \vdash \mathbf{0} : \text{end}} \text{[T-}\mathbf{0}\text{]}}{\forall i \in I \quad \Theta, x_i : B_i \vdash P_i : T_i} \text{[T-EXT]} \quad \frac{\Theta \vdash e : B \quad \Theta \vdash P : T}{\Theta \vdash \mathbf{q}\mathbf{m}(e).P : \mathbf{q}\oplus\mathbf{m}(B).T} \text{[T-OUT]}}{\Theta \vdash \sum_{i \in I} \mathbf{q}^{\mathbf{m}_i(x_i)}.P_i : \mathbf{q}\&\{\mathbf{m}_i(B_i).T_i\}_{i \in I}} \text{[T-COND]} \\
 \frac{\Theta, X : T \vdash P : T}{\Theta \vdash \mu X.P : T} \text{[T-REC]} \quad \frac{}{\Theta, X : T \vdash X : T} \text{[T-VAR]} \quad \frac{\Theta \vdash P : T \quad T \leq T'}{\Theta \vdash P : T'} \text{[T-SUB]} \\
 \frac{\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle \quad \forall i \in I \quad \vdash P_i : \Gamma(\mathbf{p}_i) \quad \vdash h_i : \Delta(-, \mathbf{p}_i) \quad \text{dom}(\Gamma) \subseteq \{\mathbf{p}_i \mid i \in I\}}{\langle \mathcal{C}; G \rangle \vdash \prod_{i \in I} (\mathbf{p}_i \triangleleft P_i \mid \mathbf{p}_i \triangleleft h_i)} \text{[T-SESS]}
 \end{array}$$

■ **Figure 9** Typing rules for queues, processes, and sessions.

► **Definition 17** (Configuration Liveness). *Given a set of reliable roles \mathcal{R} , we say that a configuration $\Gamma; \Delta$ is \mathcal{R} -live iff:*

1. $\Gamma; \Delta$ is \mathcal{R} -safe; and,
2. $\Gamma; \Delta \rightarrow_{\mathcal{R}}^* \Gamma'; \Delta'$ implies all non-crashing paths starting with $\Gamma'; \Delta'$ that are fair are also live.

A configuration $\Gamma; \Delta$ is \mathcal{R} -live when it is \mathcal{R} -safe and any reductum of $\Gamma; \Delta$ (via transition $\rightarrow_{\mathcal{R}}^*$) consistently leads to a live path if it is fair.

Properties by Projection. We conclude by showing the guarantee of safety, deadlock-freedom, and liveness in configurations associated with global types in Lem. 18. Furthermore, as a corollary, Thm. 19 demonstrates that a typing context projected from a global type (without runtime constructs) is inherently safe, deadlock-free, and live by construction.

► **Lemma 18.** *If $\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$, then $\Gamma; \Delta$ is \mathcal{R} -safe, \mathcal{R} -deadlock-free, and \mathcal{R} -live.*

► **Theorem 19** (Safety, Deadlock-Freedom, and Liveness by Projection). *Let G be a global type without runtime constructs, and \mathcal{R} be a set of reliable roles. If Γ is a typing context associated with the global type $G: \Gamma \sqsubseteq_{\mathcal{R}} G$, then $\Gamma; \Delta_{\emptyset}$ is \mathcal{R} -safe, \mathcal{R} -deadlock-free, and \mathcal{R} -live.*

5 Typing System with Crash-Stop Semantics

In this section, we present a type system for our asynchronous multiparty session calculus. Our typing system is extended from the one in [16] with crash-stop failures. We introduce the typing rules in § 5.1, and show various properties of typed sessions: subject reduction, session fidelity, deadlock-freedom, and liveness in § 5.2.

5.1 Typing Rules

Our type system uses three kinds of typing judgements: (1) for processes; (2) for queues; and (3) for sessions, and is defined inductively by the typing rules in Fig. 9. Typing judgments for processes are of form $\Theta \vdash P : T$, where Θ is a typing context for variables, defined as $\Theta ::= \emptyset \mid \Theta, x : B \mid \Theta, X : T$.

With regard to queues, we use judgments of the form $\vdash h : \delta$, where we use δ to denote a partially applied queue lookup function. We write $\delta = \Delta(-, \mathbf{p})$ to describe the incoming queue for a role \mathbf{p} , as a partially applied function $\delta = \Delta(-, \mathbf{p})$ such that $\delta(\mathbf{q}) = \Delta(\mathbf{q}, \mathbf{p})$. We write $\delta_1 \cdot \delta_2$ to denote the point-wise application of concatenation. For empty queues (ϵ), unavailable queues (\emptyset), and queue concatenations (\cdot), we simply lift the process-level queue constructs to type-level counterparts. For a singleton message $(\mathbf{q}, \mathbf{m}(v))$, the appropriate partial queue δ would be a singleton of $\mathbf{m}(B)$ (where B is the type of v) for \mathbf{q} , and an empty queue (ϵ) for any other role.

Finally, we use judgments of the form $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$ for sessions. We use a global type-guided judgment, effectively asserting that all participants in the session respect the prescribed global type, as is the case in [15]. As **highlighted**, the global type with crashed roles $\langle \mathbf{C}; G \rangle$ must have some associated configuration $\Gamma; \Delta$, used to type the processes and the queues respectively. Moreover, all the entries in the configuration must be present in the session.

Most rules in Fig. 9 assign the corresponding session type according to the behaviour of the process. For example, (**highlighted**) rule $[\text{T-}\emptyset]$ assigns the unavailable queue type \emptyset to a unavailable queue \emptyset ; rules $[\text{T-OUT}]$ and $[\text{T-EXT}]$ assign internal and external choice types to input and output processes; (**highlighted**) rule $[\text{T-}\dagger]$ (resp. $[\text{T-}\mathbf{0}]$) assigns the crash termination **stop** (resp. successful termination **end**) to a crashed process \dagger (resp. inactive process $\mathbf{0}$).

► **Example 20.** Consider the process that acts as the role \mathbf{C} in our Simpler Logging example (§2 and Ex. 14): $P_{\mathbf{C}} = \mathbf{I!read.I?report}(x).\mathbf{0}$, and a message queue $h_{\mathbf{C}} = \epsilon$. Process $P_{\mathbf{C}}$ has the type $T_{\mathbf{C}}$, and queue $h_{\mathbf{C}}$ has the type ϵ , which can be verified in the standard way. If we follow a crash reduction, e.g. by the rule $[\text{R-}\dagger]$, the session evolves as $\mathbf{C} \triangleleft P_{\mathbf{C}} \mid \mathbf{C} \triangleleft h_{\mathbf{C}} \rightarrow_{\mathcal{R}} \mathbf{C} \triangleleft \dagger \mid \mathbf{C} \triangleleft \emptyset$, where, by $[\text{T-}\dagger]$, $P_{\mathbf{C}}$ is typed by **stop**, and $h_{\mathbf{C}}$ is typed by \emptyset .

5.2 Properties of Typed Sessions

We present the main properties of typed sessions: *subject reduction* (Thm. 21), *session fidelity* (Thm. 22), *deadlock-freedom* (Thm. 24), and *liveness* (Thm. 26).

Subject reduction states that well-typedness of sessions are preserved by reduction. In other words, a session governed by a global type continues to be governed by a global type.

► **Theorem 21 (Subject Reduction).** *If $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$ and $\mathcal{M} \rightarrow_{\mathcal{R}} \mathcal{M}'$, then either $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}'$, or there exists $\langle \mathbf{C}'; G' \rangle$ such that $\langle \mathbf{C}; G \rangle \rightarrow_{\mathcal{R}} \langle \mathbf{C}'; G' \rangle$ and $\langle \mathbf{C}'; G' \rangle \vdash \mathcal{M}'$.*

Session fidelity states the opposite implication with regard to subject reduction: sessions respect the progress of the governing global type.

► **Theorem 22 (Session Fidelity).** *If $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$ and $\langle \mathbf{C}; G \rangle \rightarrow_{\mathcal{R}}$, then there exists \mathcal{M}' and $\langle \mathbf{C}'; G' \rangle$ such that $\langle \mathbf{C}; G \rangle \rightarrow_{\mathcal{R}} \langle \mathbf{C}'; G' \rangle$, $\mathcal{M} \rightarrow_{\mathcal{R}^*} \mathcal{M}'$ and $\langle \mathbf{C}'; G' \rangle \vdash \mathcal{M}'$.*

Session *deadlock-freedom* means that the “successful” termination of a session may include crashed processes and their respective unavailable incoming queues – but reliable roles (which cannot crash) can only successfully terminate by reaching inactive processes with empty incoming queues. We formalise the definition of deadlock-free sessions in Def. 23 and show that a well-typed session is deadlock-free in Thm. 24.

► **Definition 23 (Deadlock-Free Sessions).** *A session \mathcal{M} is deadlock-free iff $\mathcal{M} \rightarrow_{\mathcal{R}^*} \mathcal{M}' \not\rightarrow_{\mathcal{R}}$ implies either $\mathcal{M}' \equiv \mathbf{p} \triangleleft \mathbf{0} \mid \mathbf{p} \triangleleft \epsilon$, or $\mathcal{M}' \equiv \mathbf{p} \triangleleft \dagger \mid \mathbf{p} \triangleleft \emptyset$.*

► **Theorem 24 (Session Deadlock-Freedom).** *If $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$, then \mathcal{M} is deadlock-free.*

```

1 global protocol SimpleLogger(role U, reliable role L)
2 { rec t0 { choice at U { write(String) from U to L;
3     continue t0; }
4     or { read from U to L;
5     report(Log) from L to U;
6     continue t0; }
7     or { crash from U to L; } } }

```

■ **Figure 10** A Simple Logger protocol in SCRIBBLE.

Finally, we show that well-typed sessions guarantee the property of *liveness*: a session is *live* when all its input processes will be performed eventually, and all its queued messages will be consumed eventually. We formalise the definition of live sessions in Def. 25 and conclude by showing that a well-typed session is live in Thm. 26.

► **Definition 25** (Live Sessions). *A session \mathcal{M} is live iff $\mathcal{M} \rightarrow_{\mathcal{R}^*} \mathcal{M}' \equiv \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathcal{M}''$ implies:*

1. if $h_{\mathbf{p}} = (\mathbf{q}, \mathbf{m}(v)) \cdot h'_{\mathbf{p}}$, then $\exists P', \mathcal{M}''' : \mathcal{M}' \rightarrow_{\mathcal{R}^*} \mathbf{p} \triangleleft P' \mid \mathbf{p} \triangleleft h'_{\mathbf{p}} \mid \mathcal{M}'''$; and
2. if $P = \sum_{i \in I} \mathbf{q}^? \mathbf{m}_i(x_i).P_i$, then $\exists k \in I, w, h'_p, \mathcal{M}''' : \mathcal{M}' \rightarrow_{\mathcal{R}^*} \mathbf{p} \triangleleft P_k \{w/x_k\} \mid \mathbf{p} \triangleleft h'_p \mid \mathcal{M}'''$.

► **Theorem 26** (Session Liveness). *If $\langle \mathcal{C}; G \rangle \vdash \mathcal{M}$, then \mathcal{M} is live.*

6 Teatrino: Generating Scala Programs from Protocols

In this section, we present our toolchain TEATRINO that implements our extended MPST theory with crash-stop failures. TEATRINO processes protocols represented in the SCRIBBLE protocol description language, and generates protocol-conforming Scala code that uses the EFFPI concurrency library. A user specifies a multiparty protocol in SCRIBBLE as input, introduced in §6.1. We show the style of our generated code in §6.2, and how a developer can use the generated code to implement multiparty protocols. As mentioned in §2, generating channels for each process and type poses an interesting challenge, explained in §6.3.

6.1 Specifying a Multiparty Protocol in Scribble

The SCRIBBLE Language [43] is a multiparty protocol description language that relates closely to MPST theory (cf. [31]), and provides a programmatic way to express global types. As an example, Fig. 10 describes the following global type of a simple distributed logging protocol:

$$G = \mu t_0. \mathbf{u} \rightarrow \mathbf{l} : \{ \text{write}(\text{str}).t_0, \text{read}.\mathbf{l} \rightarrow \mathbf{u} : \text{report}(\text{Log}).t_0, \text{crash}.\text{end} \}.$$

The global type is described by a SCRIBBLE `global protocol`, with roles declared on Line 1. A transmission in the global type (e.g. $\mathbf{u} \rightarrow \mathbf{l} : \{ \dots \}$) is in the form of an interaction statement (e.g. \dots `from U to L`), except that choice (i.e. with an index set $|I| > 1$) must be marked explicitly by a `choice` construct (Line 2). Recursions and type variables in the global types are in the forms of `rec` and `continue` statements, respectively.

In order to express our new theory, we need two extensions to the language:

- (1) a reserved label `crash` to mark crash handling branches (cf. the special label `crash` in the theory), e.g. on Line 7; and
- (2) a `reliable` keyword to mark the reliable roles in the protocol (cf. the reliable role set \mathcal{R} in the theory). Roles are assumed unreliable unless declared using the `reliable` keyword, e.g. L on Line 1.

6.2 Generating Scala Code from Scribble Protocols

The Effpi Concurrency Library. [38] provides an embedded Domain Specific Language (DSL) offering a simple actor-based API. The library utilises advanced type system features in Scala 3, and provides both type-level and value-level constructs for processes and channels. In particular, the type-level constructs reflect the behaviour of programs (i.e. processes), and thus can be used as specifications. Following this intuition, we generate process types that reflect local types from our theory, as well as a tentative process implementing that type (by providing some default values where necessary).

Generated Code. To illustrate our approach, we continue with the Simple Logger example from § 6.1, and show the generated code in Fig. 11. The generated code can be divided into five sections:

- (i) label and payload declarations,
- (ii) recursion variable declarations,
- (iii) local type declarations,
- (iv) role-implementing functions, and
- (v) an entry point.

Sections (i) and (ii) contain boilerplate code, where we generate type declarations for various constructs needed for expressing local types and processes. We draw attention to the *key* sections (iii) and (iv), where we generate a representation of local types for each role, as well as a tentative process inhabiting that type.

Local Types and Effpi Types. We postpone the discussion about channels in EFFPI to § 6.3. For now, we compare the generated EFFPI type and the projected local type, and also give a quick primer³ on EFFPI constructs. The projected local types of the roles **u** and **l** are shown as follows:

$$\begin{aligned} G \upharpoonright_{\{1\}} \mathbf{u} &= \mu t_0. \mathbf{l} \oplus \{ \text{write}(\text{str}).t_0, \text{read}.\mathbf{l} \& \text{report}(\text{Log}).t_0 \} \\ G \upharpoonright_{\{1\}} \mathbf{l} &= \mu t_0. \mathbf{u} \& \{ \text{write}(\text{str}).t_0, \text{read}.\mathbf{u} \oplus \text{report}(\text{Log}).t_0, \text{crash}.\text{end} \} \end{aligned}$$

The local types are recursive, and the EFFPI type implements recursion with `Rec[RecT0, ...]` and `Loop[RecT0]`, using the recursion variable `RecT0` declared in section (ii).

For role **u**, The inner local type is a sending type towards role **l**, and we use an EFFPI process output type `Out[A, B]`, which describes a process that uses a channel of type **A** to send a value of type **B**. For each branch, we use a separate output type, and connect it to the type of the continuation using a sequential composition operator (`>>:`). The different branches are then composed together using a union type (`|`) from the SCALA 3 type system.

Recall that the role **l** is declared `reliable`, and thus the reception labelled `report` from **l** at **u** does not need to contain a crash handler. We use an EFFPI process input type `In[A, B, C]`, which describes a process that uses a channel of type **A** to receive a value of type **B**, and uses the received value in a continuation of type **C**.

For role **l**, the reception type is more complex for two reasons:

- (1) role **u** is unreliable, necessitating crash handling; and
- (2) the reception contains branching behaviour (cf. the reception **u** being a singleton), with labels `write` and `read`.

³ A more detailed description of constructs can be found in [39].

```

1 // (i) label and payload declarations
2 case class Log() // payload type
3 case class Read() // label types
4 case class Report(x : Log)
5 case class Write(x : String)
6 // (ii) recursion variable declarations
7 sealed abstract class RecT0[A]() extends RecVar[A]("RecT0")
8 case object RecT0 extends RecT0[Unit]
9 // (iii) local type declarations
10 type U[C0 <: OutChan[Read | Write], C1 <: InChan[Report]] =
11   Rec[RecT0,
12     ( (Out[C0, Read] >>: In[C1, Report, (x0 : Report) => Loop[RecT0]]
13       | (Out[C0, Write] >>: Loop[RecT0]) ) ]
14
15 type L[C0 <: InChan[Read | Write], C1 <: OutChan[Report]] =
16   Rec[RecT0,
17     InErr[C0, Read | Write, (x0 : Read | Write) => L0[x0.type, C1],
18           (err : Throwable) => PNil]]
19
20 type L0[X0 <: Read | Write, C1 <: OutChan[Report]] <: Process =
21   X0 match { case Read => Out[C1, Report] >>: Loop[RecT0]
22             case Write => Loop[RecT0] }
23 // (iv) role-implementing functions
24 def u(c0 : OutChan[Read | Write],
25     c1 : InChan[Report]) : U[c0.type, c1.type] = {
26   rec(RecT0) {
27     val x0 = 0
28     if (x0 == 0) {
29       send(c0, new Read()) >> receive(c1) {(x1 : Report) => loop(RecT0) }
30     } else {
31       send(c0, new Write("")) >> loop(RecT0)
32     } } }
33
34 def l(c0 : InChan[Read | Write],
35     c1 : OutChan[Report]) : L[c0.type, c1.type] =
36   rec(RecT0) {
37     receiveErr(c0)((x0 : Read | Write) => l0(x0, c1),
38                 (err : Throwable) => nil) }
39
40 def l0(x : Read | Write, c1 : OutChan[Report]) : L0[x.type, c1.type] =
41   x match { case y : Read => send(c1, new Report(new Log())) >> loop(RecT0)
42           case y : Write => loop(RecT0) }
43 // (v) an entry point (main object)
44 object Main {
45   def main() : Unit = {
46     var c0 = Channel[Read | Write]()
47     var c1 = Channel[Report]()
48     eval(par(u(c0, c1), l(c0, c1)))
49   } }

```

■ **Figure 11** Generated SCALA code for the Simple Logger protocol in Fig. 10.

For (1), we extend EFFPI with a variant of the input process type `InErr[A, B, C, D]`, where `D` is the type of continuation in case of a crash. For (2), the payload type is first received as an union (Line 17), and then `matched` to select the correct continuation according to the type (Line 21).

From Types To Implementations. Since EFFPI type-level and value-level constructs are closely related, we can easily generate the processes from the processes types. Namely, by matching the type `Out[...]` with the process `send(...)`; the type `In[...]` with the process `receive(...)` `{... => ...}`; and similarly for other constructs. Whilst executable, the generated code represents a skeleton implementation, and the programmer is expected to alter the code according to their requirements.

We also introduce a new crash handling receive process `receiveErr`, to match the new `InErr` type. Process crashes are modelled by (caught) exceptions and errors in role-implementing functions, and crash detection is achieved via timeouts. Timeouts are set by the programmer in an (implicit) argument to each `receiveErr` call.

Finally, the entry point (main object) in section (v) composes the role-implementing functions together with `par` construct in EFFPI, and connects the processes with channels.

6.3 Generating Effpi Channels from Scribble Protocols

As previously mentioned, EFFPI processes use *channels* to communicate, and the type of the channel is reflected in the type of the process. However, our local types do not have any channels; instead, they contain a partner role with which to communicate. This poses an interesting challenge, and we explain the channel generation procedure in this section.

We draw attention to the generated code in Fig. 11 again, where we now focus on the parameters `C0` in the generated types `U` and `L`. In the type `U`, the channel type `C0` needs to be a subtype of `OutChan[Read | Write]` (Line 10), and we see the channel is used in the output processes types, e.g. `Out[C0, Read]` (Line 12, note that output channels subtyping is covariant on the payload type). Dually, in the type `L`, the channel type `C0` needs to be a subtype of `InChan[Read | Write]` (Line 15), and we see the channel is used in the input process type, i.e. `InErr[C0, Read | Write, ..., ...]` (Line 17).

Similarly, a channel `c0` is needed in the role-implementing functions `u` and `l` as arguments, and the channel is used in processes `send(c0, ...)` and `receiveErr(c0) ...`. Finally, in the entry point, we create a bidirectional channel `c0 = Channel[Read | Write]()` (Line 46), and pass it as an argument to the role-implementing functions `u` and `l` (Line 48), so that the channel can be used to link two role-implementing processes together for communication.

Generating the channels correctly is crucial to the correctness of our approach, but non-trivial since channels are implicit in the protocols. In order to do so, a simple approach is to traverse each interaction in the global protocol, and assign a channel to each accordingly.

This simple approach would work for the example we show in Fig. 10; however, it would not yield the correct result when *merging* occurs during projection, which we explain using an example. For clarity and convenience, we use *annotated* global and local types, where we assign an identifier for each interaction to signify the channel to use, and consider the following global type: $G = p \xrightarrow{0} q: \left\{ \text{left.} p \xrightarrow{1} r: \text{left.end}, \text{right.} p \xrightarrow{2} r: \text{right.end} \right\}$.

The global type describes a simple protocol, where role `p` selects a label `left` or `right` to `q`, and `q` passes on the same label to `r`. As a result, the projection on `r` (assuming all roles reliable) should be a reception from `q` with branches labelled `left` or `right`, i.e. $p \&^{1,2} \{ \text{left.end}, \text{right.end} \}$. Here, we notice that the interaction between `q` and `r` should take place on a single channel, instead of two separate channels annotated 1 and 2.

■ **Table 1** Overview of All Variants for Each Example.

Name	Var.	\mathcal{R}	Comms.	Crash Branches	Max Cont. Len.
PingPong $\mathcal{R} = \{p, q\}$	(a)	\mathcal{R}	2	0	4
	(b)	\emptyset	2	2	4
Adder $\mathcal{R} = \{p, q\}$	(c)	\mathcal{R}	5	0	6
	(d)	\emptyset	5	5	6
TwoBuyer $\mathcal{R} = \{p, q, r\}$	(e)	\mathcal{R}	7	0	8
	(f)	$\{r\}$	18	6	12
OAuth $\mathcal{R} = \{c, a, s\}$	(g)	\mathcal{R}	12	0	11
	(h)	$\{s, a\}$	21	8	11
	(i)	$\{s\}$	26	13	11
	(j)	\emptyset	30	28	11
TravelAgency $\mathcal{R} = \{c, a, s\}$	(k)	\mathcal{R}	8	0	6
	(l)	$\{a, s\}$	9	3	6
	(m)	$\{a\}$	9	4	6
DistLogger $\mathcal{R} = \{l, c, i\}$	(n)	\mathcal{R}	10	0	7
	(o)	$\{i, c\}$	15	2	7
	(p)	$\{i\}$	16	4	7
CircBreaker $\mathcal{R} = \{s, a, r\}$	(q)	\mathcal{R}	18	0	10
	(r)	$\{a, s\}$	24	3	10
	(s)	$\{a, s\}$	23	3	11

When *merging* behaviour occurs during projection, we need to use the same channel in those interactions to achieve the correct behaviour. After traversing the global type to annotate each interaction, we merge annotations involved in merges during projection.

7 Evaluation

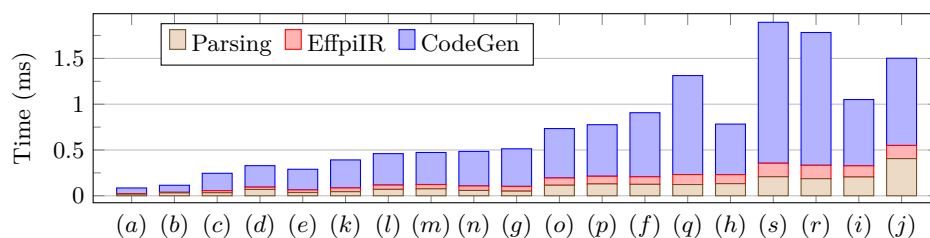
We evaluate our toolchain TEATRINO from two perspectives: *expressivity* and *feasibility*. For expressivity, we use examples from session type literature, and extend them to include crash handling behaviour using two patterns: failover and graceful failure. For feasibility, we show that our tool generates SCALA code within negligible time.

We note that we do *not* evaluate the performance of the generated code. The generated code uses the EFFPI concurrency library to implement protocols, and any performance indication would depend and reflect on the performance of EFFPI, instead of TEATRINO.

Expressivity. We evaluate our approach with examples in session type literature: PingPong, Adder, TwoBuyer [21], OAuth [32], TravelAgency [23], DistLogger [26], and CircBreaker [26]. Notably, the last two are inspired by real-world patterns in distributed computing.

We begin with the *fully reliable* version of the examples, and extend them to include crash handling behaviour. Recall that our extended theory subsumes the original theory, when all roles are assumed *reliable*. Therefore, the fully reliable versions can act both as a sanity check, to ensure the code generation does not exclude good protocols in the original theory, and as a baseline to compare against.

To add crash handling behaviour, we employ two patterns: *failover* and *graceful failure*. In the former scenario, a crashed role has its functions taken over by another role, acting as a substitute to the crashed role [3]. In the latter scenario, the protocol is terminated peacefully, possibly involving additional messages for notification purposes. Using the example from §2, the fully reliable protocol in Eq. (1) is extended to one with graceful failure in Eq. (2).



■ **Figure 12** Average Generation Times for All Variants in Table 1.

We show a summary of the examples in Table 1. For each example, we give the set of all roles \mathcal{R} and vary the set of reliable roles (\mathcal{R}). Each variant is given an identifier (Var.), and each example always has a fully reliable variant where $\mathcal{R} = \mathcal{R}$. We give the number of communication interactions (Comms.), the number of `crash` branches added (Crash Branches), and the length of the longest continuation (Max Cont. Len.) in the given global type.

The largest of our examples in terms of concrete interactions is `OAuth`, with Variant (i) having 26 interactions and (j) having 30 interactions. This represents a $2.17\times$ and $2.5\times$ increase over the size of the original protocol, and is a consequence of the confluence of two factors: the graceful failure pattern, and low degree of branching in the protocol itself. The `TwoBuyer` Variant (f) represents the greatest increase ($2.57\times$) in interactions, a result of implementing the failover pattern. The `CircBreaker` variants are also notable in that they are large in terms of both interactions and branching degree – both affect generation times.

Feasibility. In order to demonstrate the feasibility of our tool `TEATRINO`, we give generation times using our prototype for all protocol variants and examples, plotted in Fig. 12. We show that `TEATRINO` is able to complete the code generation within milliseconds, which does not pose any major overhead for a developer.

In addition to total generation times, we report measurements for three main constituent phases of `TEATRINO`: parsing, `EffpiIR` generation, and code generation. `EffpiIR` generation projects and transforms a parsed global type into an intermediate representation, which is then used to generate concrete `SCALA` code.

For all variants, the code generation phase is the most expensive phase. This is likely a consequence of traversing the given `EffpiIR` representation of a protocol twice – once for local type declarations and once for role-implementing functions.

8 Related Work

We summarise related work on both theory and implementations of session types with failure handling, as well as other MPST implementations targeting `SCALA` without failures.

We first discuss closest related work [3, 27, 33, 42], where multiparty session types are extended to model crashes or failures. Both [33] and [27] are exclusively theoretical.

[33] proposes an MPST framework to model fine-grained unreliability: each transmission in a global type is parameterised by a reliability annotation, which can be one of unreliable (sender/receiver can crash, and messages can be lost), weakly reliable (sender/receiver can crash, messages are not lost), or reliable (no crashes or message losses). [42] utilises MPST as a guidance for fault-tolerant distributed system with recovery mechanisms. Their framework includes various features, such as sub-sessions, event-driven programming, dynamic role assignments, and, most importantly, failure handling. [3] develops a theory of multiparty

session types with crash-stop failures: they model crash-stop failures in the semantics of processes and session types, where the type system uses a model checker to validate type safety. [27] follow a similar framework to [3]: they model an asynchronous semantics, and support more patterns of failure, including message losses, delays, reordering, as well as link failures and network partitioning. However, their typing system suffers from its genericity, when type-level properties become undecidable [27, §4.4].

Other session type works on modelling failures can be briefly categorised into two: using affine types or exceptions [14, 26, 29], and using coordinators or supervision [1, 41]. The former adapts session types to an *affine* representation, in which endpoints may cease prematurely; the latter, instead, are usually reliant on one or more *reliable* processes that *coordinate* in the event of failure. The works [1, 29, 41] are limited to theory.

[29] first proposes the affine approach to failure handling. Their extension is primarily comprised of a *cancel operator*, which is semantically similar to our crash construct: it represents a process that has terminated early. [14] presents a concurrent λ -calculus based on [29], with asynchronous session-typed communication and exception handling, and implements their approach as parts of the LINKS language. [26] proposes a framework of *affine* multiparty session types, and provides an implementation of affine MPST in the RUST programming language. They utilise the affine type system and `Result` types of RUST, so that the type system enforces that failures are handled.

Coordinator model approaches [1, 41] often incorporate *interrupt blocks* (or similar constructs) to model crashes and failure handling. [1] extends the standard MPST syntax with *optional blocks*, representing regions of a protocol that are susceptible to communication failures. In their approach, if a process P expects a value from an optional block which fails, then a *default value* is provided to P , so P can continue running. This ensures termination and deadlock-freedom. Although this approach does not feature an explicit reliable coordinator process, we describe it here due to the inherent coordination required for multiple processes to start and end an optional block. [41] similarly extends the standard global type syntax with a *try-handle* construct, which is facilitated by the presence of a reliable coordinator process, and via a construct to specify reliable processes. When the coordinator detects a failure, it broadcasts notifications to all remaining live processes; then, the protocol proceeds according to the failure handling continuation specified as part of the try-handle construct.

Other related MPST implementations include [9, 17, 18]. [18] designs a framework for MPST-guided, safe actor programming. Whilst the MPST protocol does not include any failure handling, the actors may fail or raise exceptions, which are handled in a similar way to what we summarise as the affine technique. [9] revisits API generation techniques in SCALA for MPST. In addition to the traditional local type/automata-based code generation [22, 36], they propose a new technique based on sets of pomsets, utilising SCALA 3 match types [4]. [17] presents CHORAL, a programming language for choreographies (multiparty protocols). CHORAL supports the handling of *local exceptions* in choreographies, which can be used to program reliable channels over unreliable networks, supervision mechanisms, *etc.* for fallible communication. They utilise automatic retries to implement channel APIs.

9 Conclusion and Future Work

To overcome the challenge of accounting for failure handling in distributed systems using session types, we propose TEATRINO, a code generation toolchain. It is built on asynchronous MPST with crash-stop semantics, enabling the implementation of multiparty protocols that are resilient to failures. Desirable global type properties such as deadlock-freedom, protocol

conformance, and liveness are preserved by construction in typed processes, even in the presence of crashes. Our toolchain TEATRINO, extends SCRIBBLE and EFFPI to support crash detection and handling, providing developers with a lightweight way to leverage our theory. The evaluation of TEATRINO demonstrates that it can generate SCALA code with minimal overhead, which is made possible by the guarantees provided by our theory.

This work is a new step towards modelling and handling real-world failures using session types, bridging the gap between their theory and applications. As future work, we plan to study different crash models (e.g. crash-recover) and failures of other components (e.g. link failures). These further steps will contribute to our long-term objective of modelling and type-checking well-known consensus algorithms used in large-scale distributed systems.

References

- 1 Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2017. doi:10.1007/978-3-319-60225-7_1.
- 2 Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Designing Asynchronous Multiparty Protocols with Crash-Stop Failures, 2023. arXiv:2305.06238.
- 3 Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. In Bartek Klin, Sławomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR 2022)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:25, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2022.35.
- 4 Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proc. ACM Program. Lang.*, 6(POPL):1–24, 2022. doi:10.1145/3498698.
- 5 Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 Toolset for Analysing Concurrent Systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing.
- 6 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.
- 7 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019. doi:10.1145/3290342.
- 8 Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, March 1996. doi:10.1145/226643.226647.
- 9 Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/opus/volltexte/2022/16255>.
- 10 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693 of *PPoPP '22*, pages 261–246. ACM, 2022. doi:10.1145/3503221.3508404.

- 11 Romain Demangeon, Kohei Honda, Raymond Hu, Romyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.
- 12 Romain Demangeon and Nobuko Yoshida. On the Expressiveness of Multiparty Sessions. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*, volume 45 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 560–574, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSTTCS.2015.560.
- 13 Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 174–186, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39212-2_18.
- 14 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.
- 15 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebraic Methods Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.
- 16 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434297.
- 17 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. arXiv:2005.09520.
- 18 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2021.10.
- 19 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/BFb0053567.
- 20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284. ACM, 2008. doi:10.1145/1328897.1328472.
- 21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63:1–67, 2016. doi:10.1145/2827695.
- 22 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering*, volume 9633 of *LNCS*, pages 401–418, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49665-7_24.
- 23 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008. doi:10.1007/978-3-540-70592-5_22.
- 24 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 146–159. ACM, 2016. doi:10.1145/2967973.2968595.

- 25 Nicolas Lagaillardie, Romyana Neykova, and Nobuko Yoshida. Implementing Multiparty Session Types in Rust. In Simon Bludze and Laura Bocchi, editors, *Coordination Models and Languages – 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2020. doi:10.1007/978-3-030-50029-0_8.
- 26 Nicolas Lagaillardie, Romyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/opus/volltexte/2022/16232>.
- 27 Matthew Alan Le Brun and Ornela Dardha. $MAG\pi$: Types for Failure-Prone Communication. In Thomas Wies, editor, *Programming Languages and Systems*, pages 363–391, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30044-8_14.
- 28 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types. In *International Conference on Compiler Construction*, CC, pages 94–106, 2021. doi:10.1145/3446804.3446854.
- 29 Dimitris Mostrous and Vasco T. Vasconcelos. Affine Sessions. *Logical Methods in Computer Science*, Volume 14, Issue 4, November 2018. doi:10.23638/LMCS-14(4:14)2018.
- 30 Romyana Neykova and Nobuko Yoshida. Multiparty Session Actors. *Logical Methods in Computer Science*, 13:1–30, 2017. doi:10.23638/LMCS-13(1:17)2017.
- 31 Romyana Neykova and Nobuko Yoshida. *Featherweight Scribble*, pages 236–259. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-21485-2_14.
- 32 Romyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40787-1_25.
- 33 Kirstin Peters, Uwe Nestmann, and Christoph Wagner. Fault-tolerant multiparty session types. In Mohammad Reza Mousavi and Anna Philippou, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 93–113, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-08679-3_7.
- 34 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
- 35 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. doi:10.1017/CB09780511777110.
- 36 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2017.24.
- 37 Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. doi:10.1145/3290343.
- 38 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying Message-Passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 502–516, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3322484.
- 39 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Effpi: verified message-passing programs in Dotty. In Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom, editors, *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, pages 27–31. ACM, 2019. doi:10.1145/3337932.3338812.
- 40 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 – July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470531.

1:30 Designing Asynchronous Multiparty Protocols with Crash-Stop Failures

- 41 Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 799–826, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-89884-1_28.
- 42 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485501.
- 43 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *8th International Symposium on Trustworthy Global Computing – Volume 8358*, TGC 2013, pages 22–41, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-319-05119-2_3.

Nested Pure Operation-Based CRDTs

Jim Bauwens ✉

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Elisa Gonzalez Boix ✉

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Abstract

Modern distributed applications increasingly replicate data to guarantee high availability and optimal user experience. Conflict-free Replicated Data Types (CRDTs) are a family of data types specially designed for highly available systems that guarantee some form of eventual consistency. Designing CRDTs is very difficult because it requires devising designs that guarantee convergence in the presence of conflicting operations. Even though design patterns and structured frameworks have emerged to aid developers with this problem, they mostly focus on statically structured data; nesting and dynamically changing the structure of a CRDT remains to be an open issue.

This paper explores support for nested CRDTs in a structured and systematic way. To this end, we define an approach for building nested CRDTs based on the work of pure operation-based CRDTs, resulting in *nested pure operation-based CRDTs*. We add constructs to control the nesting of CRDTs into a pure operation-based CRDT framework and show how several well-known CRDT designs can be defined in our framework. We provide an implementation of nested pure operation-based CRDTs as an extension to the Flec, an existing TypeScript-based framework for pure operation-based CRDTs. We validate our approach, 1) by implementing a portfolio of nested data structures, 2) by implementing and verifying our approach in the VeriF_x language, and 3) by implementing a real-world application scenario and comparing its network usage against an implementation in the closest related work, Automerger. We show that the framework is general enough to nest well-known CRDT designs like maps and lists, and its performance in terms of network traffic is comparable to the state of the art.

2012 ACM Subject Classification Software and its engineering → Consistency; Computer systems organization → Distributed architectures; Software and its engineering → Synchronization; Software and its engineering → Middleware; Software and its engineering → Reflective middleware

Keywords and phrases CRDTs, replication, pure operation-based CRDTs, composition, nesting

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.2

Supplementary Material *Software (Source Code)*: <https://gitlab.soft.vub.ac.be/jimbauwens/flec>

Funding *Jim Bauwens*: Fonds Wetenschappelijk Onderzoek - Vlaanderen: FWOSB90

1 Introduction

To ease the development of geo-distributed applications, much research has studied the concept of *replicated data types* (RDTs). An RDT exposes to programmers an interface akin to that of a sequential data type while incorporating mechanisms to keep data consistent across replicas [9, 22, 14]. Conflict-Free Replicated Data Types [22, 21, 19] (CRDTs) are the most well-known family of replicated data types. CRDTs guarantee strong eventual consistency (SEC) [22] that adds to eventual consistency the guarantee of *state convergence*, i.e. if two replicas of the data type have received the same updates, they will be in the same state. This implies that replicas converge without synchronisation or conflicts because they reach the same state as soon as they have observed the same operations.



© Jim Bauwens and Elisa Gonzalez Boix;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 2; pp. 2:1–2:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2:2 Nested Pure Operation-Based CRDTs

Designing new RDTs that guarantee convergence is a complex task. Only for data types for which all operations commute (e.g., counters), one can easily construct a CRDT (since regardless of the ordering in which operations are applied, the resulting state will be equivalent). A common approach to designing CRDTs is to use causal ordering for non-concurrent operations and handle conflicts between non-commutative concurrent operations [19, 13, 3]. Many current designs handle those conflicts in an ad-hoc way crafted for each data type, often relying on specific meta-data to track causality and relations. For example, some CRDT designs (e.g., OR-Set) use *tombstones* to ensure that removal operations commute [22]. However, for many CRDT types, this meta-data grows unboundedly. Moreover, it is very difficult to modify existing designs (e.g., add operations to the data type, or modify the design to work with different networking assumptions). Pure operation-based CRDTs [3] aim to solve those issues and propose an approach for building operation-based CRDTs based on a Partial Ordered Log (PO-Log) of operations. The approach exposes causal information from the underlying communication middleware which can be used to enable the removal of redundant meta-data. While pure operation-based CRDTs provide a structured framework to build CRDTs, it is designed to build CRDTs for flat data structures.

In this work, we focus on the issues raised by composing CRDTs, e.g., when CRDTs are nested or more than one CRDT is combined into a new one. Composing CRDTs is non-trivial, as the convergence property of a CRDT design is made to hold for a *single* CRDT but does not necessarily hold when several CRDTs are composed into a new one. Recent work has explored what concurrency semantics can be utilised for composing designs [19] and several specific implementations exist [15, 16, 18]. Existing approaches, however, mainly follow a state-based design, in which any information on applied operations is lost during the merging process. This may result in non-sensible designs for nested CRDTs and hampers the development of CRDTs where the operation history needs to be used to improve the merging algorithm. For example, recent work [25] explores the design of a distributed file system CRDT that uses nested structures for storing filesystem metadata. They argue that to properly support authentication primitives, all semantically related authentication information needs to be combined and considered in the merging semantics.

Operation-based techniques, on the other hand, are better suited for replicating nested data structures as information on applied operations can be used to determine the optimal ordering for concurrent operations. In the context of nested structures, this means that it is less complex to relate different operations or even separate them when deciding what nested semantics for non-commutative concurrent operations are needed. To the best of our knowledge, no uniform (structured) approach exists for designing and implementing nested CRDTs, where CRDT designers can easily coordinate the interaction between nested structures, as part of the concurrency semantics of the replicated structure. In this paper, we introduce a general approach to nesting and composing pure operation-based CRDTs and propose a framework for implementing pure operation-based nested CRDTs. For this, we extend the pure operation-based CRDT framework [3] with support for nested CRDT structures. We implement our approach by extending an existing pure operation-based CRDT framework written in TypeScript called Flec [5], where we develop a portfolio of nested data structures. We demonstrate the correctness of our approach using a VeriFy implementation where we verify that the structures always remain strong eventually consistent. Finally, we implement a distributed file system based on Vanakieva et al. [25] to assess the performance of our approach in comparison to a state-of-the-art JSON CRDT implementation, Automerge [15].

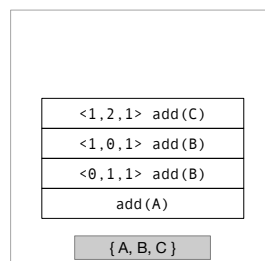
To summarise, we introduce the following contributions:

- A general approach for the design and implementation of nested CRDTs, building on the work of pure operation-based CRDTs.
- A full-fledged TypeScript implementation of our approach which includes a portfolio of existing and novel pure operation-based CRDTs.
- A validation of the correctness of our nested pure operation-based framework and a portfolio of CRDTs built on this framework.
- A performance evaluation showing that our approach has reduced network usage when compared to Automerge [15].

2 Background

In this section, we provide the necessary background to understand the contributions of this work. Baquero et al. [2] introduced the pure operation-based framework for designing CRDTs in a structured way while avoiding performance issues related to the unbound growth of meta-data. They build on the idea of relying on Reliable Causal Broadcast [8] (RCB) middleware to ensure causal ordering for non-concurrent operations (along with reliable delivery) [22, 2]. Instead of manually encoding causality information as meta-data to operations, the framework exposes causality information stored within the RCB middleware to CRDT implementors. More concretely, the framework employs a partially ordered log of operations (PO-Log) constructed with the causality information of the underlying RCB middleware. The state of the data structure can be computed by observing this log, and the log can be compacted to ensure that memory does not grow unboundedly. Figure 1 shows an example of a PO-Log of an Add-Wins (AW-Set) set replica (in a system of three replicas). It contains four add operations, which form the state $\{A, B, C\}$, depicted in grey. Three of these operations include causality information from the underlying RCB middleware, i.e. they carry a vector clock.

Algorithm 1 shows the distributed algorithm describing the interaction between the RCB middleware and the pure operation-based CRDT framework. Each replica contains has a particular state (s_i for replica i), representing its PO-Log. The *operation*(o) method is called by client applications (e.g. by a CRDT implementation using the pure operation-based framework) when an operation o should be applied. It ensures that operations are broadcasted to other replicas and annotated with a logical timestamp on delivery (t in the algorithm description). It does this by invoking the *broadcast* method from the RCB layer, which broadcasts the operation with the associated timestamp meta-data to all other replicas. On delivery of these operations (and after all causal dependencies are met), the RCB layer will invoke the *deliver*(t, o) method from the pure operation-based framework, where the log (s_i) will be modified if needed.



■ **Figure 1** The internal state of an AW-Set. One operation is causally stable, and as such does not contain a timestamp. Together, the operations form the state $\{A, B, C\}$.

2:4 Nested Pure Operation-Based CRDTs

The framework introduces the concept of *causal redundancy* to keep the log compact. The idea is that a particular operation may make existing operations in the log redundant, or that the arriving operation may be redundant itself. Rules for this can be defined by using two binary redundancy relations, \mathbf{R} and \mathbf{R}_- . \mathbf{R}_- defines whether an arriving operation makes existing entries in the log redundant, and \mathbf{R} defines if a newly arriving operation should be stored in the log. The definitions for these relations need to be provided by the concrete CRDT implementation. The framework can also determine when operations are *causally stable*, i.e., they have been observed on all replicas, and trim causal information for their log entries. Since new operations can never be concurrent with causally stable operations, their causal meta-data (such as timestamps) is thus no longer needed. The RCB layer can determine causal stability by comparing the vector clocks of incoming messages and decide whether a particular timestamp must have been observed by all nodes. Whenever a particular timestamp is causally stable, the `stable` function will be invoked by the RCB layer, and the framework will compact stable operations that are returned by the `stabilize` function. It does this by replacing (removing) the associated timestamp with the bottom (null) element. This can also be seen in Figure 1, where the `add(A)` operation has been stripped from causality information. Similarly to the redundancy relations, the `stabilize` function has to be provided by any CRDT implementation built on the framework.

■ **Algorithm 1** (Simplified) distributed algorithm for a replica i showing the interaction between the RCB middleware and the pure op-based CRDT framework.

```

state:  $s_i := \emptyset$ 
on  $operation_i(o)$  :
  |  $broadcast_i(o)$ 
on  $deliver_i(t, o)$  :
  |  $s_i := (s_i \setminus \{(t', o') \mid (t', o') \in s_i \cdot (t', o') \mathbf{R}_-(t, o)\}) \cup \{(t, o) \mid (t, o) \not\mathbf{R} s_i\}$ 
on  $stable_i(t)$  :
  |  $stabilize_i(t, s_i)[(\perp, o)/(t, o)]$ 

```

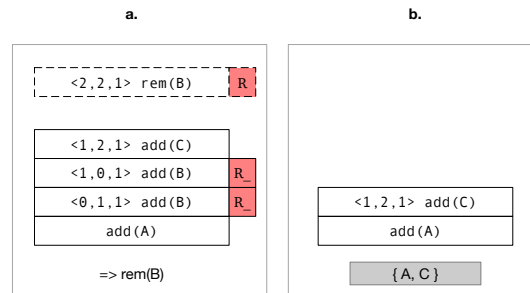
Table 1 shows the implementation for an AW-Set CRDT in the pure operation-based framework. The table is grouped as follows: (1) functions that are used by the framework and that dictate the interaction between new operations and entries in the log, and (2) procedures that can be invoked by the user for state serialisation or mutations.

The \mathbf{R} relation for the add-wins set defines that the `clear` and `remove` operations will never be stored in the log. \mathbf{R}_- , on the other hand, defines that an arriving operation o will make any stored operations (in the log) redundant if and only if the stored operation o' causally happened before the arriving operation (i.e. $t' < t$) and the arriving operation is acting on the same set element, or the arriving operation is a `clear` (i.e., which removes all happened-before elements). For example, a `remove(X)` will make a previous `add(X)` redundant; and a `clear` operation will remove all previous log entries. The combination of both rules ensures that `add` operations will always 'win' from concurrent operations. The implementation of `stabilize` defines that all causally stable operations will be stripped from their timestamps (to preserve memory consumption). Additionally, the log will only contain distinct `add` operations at any point in time. To query the state, a map function can extract each element from these operations (as shown in the `toList` function) and serialise it into an actual set data structure.

Figure 2 illustrates the internal state and the PO-Log of the AW-Set depicted in Figure 1 after receiving a `remove(B)` operation (depicted in the a. box) and after the operation has been applied (depicted in the b. box). Initially, the log consists of an operation which is

■ **Table 1** Semantics for the add-wins pure-op set, based on the approach in [3].

Pure	$(t, o) \mathbf{R} s = \text{op}(o) = (\text{clear} \vee \text{remove})$ $(t', o') \mathbf{R}_- (t, o) = t' < t \wedge (\text{op}(o) = \text{clear} \vee \text{arg}(o) = \text{arg}(o'))$ $\text{stabilize}(t, s) = s$
User	$\text{toList}(s) = \{v \mid (_, [\text{op}=\text{add}, \text{arg}=v]) \in s\}$ $\text{add}(e) = \text{operation}([\text{op}=\text{add}, \text{arg}=e])$ $\text{remove}(e) = \text{operation}([\text{op}=\text{remove}, \text{arg}=e])$



■ **Figure 2** The internal states of an AW-Set, after receiving a remove (rem) operation, and after the operation has been applied.

causally stable (the **add(a)**), and three other operations which are not yet stable. Looking at the vector clocks, we can observe that the log has two concurrent operations, both of which add element B. When the arriving **remove(B)** is checked against these stored operations, both previous **add(B)** operations will be marked as redundant by the **R₋** relation (as the operations have the same key, and are causal predecessors). Additionally, the arriving operation itself is immediately marked as redundant by the **R** relation of the AW-Set semantics (all **remove** and **clear** operations are immediately redundant) and as such, it will not be added to the log. The box denoted by *b*. shows the final result of applying **remove(B)**: no entries for adding element B remain, and the removal operation itself was not added to the log. Thus, the replica state becomes $\{A, C\}$.

3 Nesting Pure Operation-Based CRDTs

Currently, it is not possible to reason about nested structures within the pure operation-based CRDT framework. Redundancy relations only work on a flat level, and any logic to traverse hierarchical/nested structures would have to be manually bolted on top of the framework in an ad-hoc way.

As there is no native support for this functionality, nested designs built with the current framework require developers to store nested operations in a flattened form in the main log. To evaluate and apply the contents of the log, developers would need to either fully combine the logic of the nested and main top-level CRDT or encode the nested CRDT semantics in the query functions. In the former case, the redundancy relations and query functions would have to manage all concurrency rules for all needed nested strategies. This greatly complicates the design of such structures and makes them more prone to errors. In the latter case, only the query functions would need to be touched, but they would have to implement all redundancy logic from scratch. A programmer could delegate operations to separate components for the nested CRDTs, but in the end, this implies a reimplementaion of the delivery of operations in the query function logic while this should be kept in the framework.

In this work, we propose a novel nested pure operation-based CRDT framework that enables the systematic construction of nested data structures building on the ideas of Baquero et al [2]. We explore a framework that allows developers to combine and nest existing pure operation-based CRDTs and provides constructs for the development of novel CRDTs. In particular, we focus on designs where nested structures can dynamically change at runtime, i.e., data structures that grow and shrink during the lifetime of an application, such as maps and lists, where values can be CRDTs. Our approach offers developers novel framework constructs to define the relationship between parent and child CRDT. The framework then handles all replication aspects regarding the delivery of operations in the data-structure hierarchy, ensuring that causal ordering is respected and that nested children are recursively reset when needed. In the following section, we will focus on the CRDT framework level and detail our extensions to pure operation-based CRDTs to support nesting.

3.1 Extending the Pure Operation-Based Framework

In this work, we model a nested data structure as a nested hierarchy where children can be identified by a particular key and deeply nested children by an absolute path (list of keys) relative to the topmost data structure (the root CRDT). To support nested data structures, we introduce three extensions to the pure operation-based framework:

- An internal data structure to keep track of nested CRDTs (i.e., the *children* of a CRDT).
- An update propagation mechanism for nested CRDTs that delivers the applied operations ensuring that the concurrency semantics of parent data structures are upheld.
- A reset mechanism for nested CRDT operations that ensures that the concurrency semantics of children's data structures are upheld.

Each of these extensions is essential to ensure the correctness of replicated data types. In the following sections, we elaborate on them and motivate why they are needed.

3.1.1 Keeping Track of Nested Data Structures

Objects or data structures that have nested children typically refer to children by some key. Our approach assumes that children have a unique identifier by which they can be accessed (i.e., queried and updated). As nested children can also contain other nested elements, an absolute path can be constructed to identify a particular nested data structure, starting from the root (top-most) data structure.

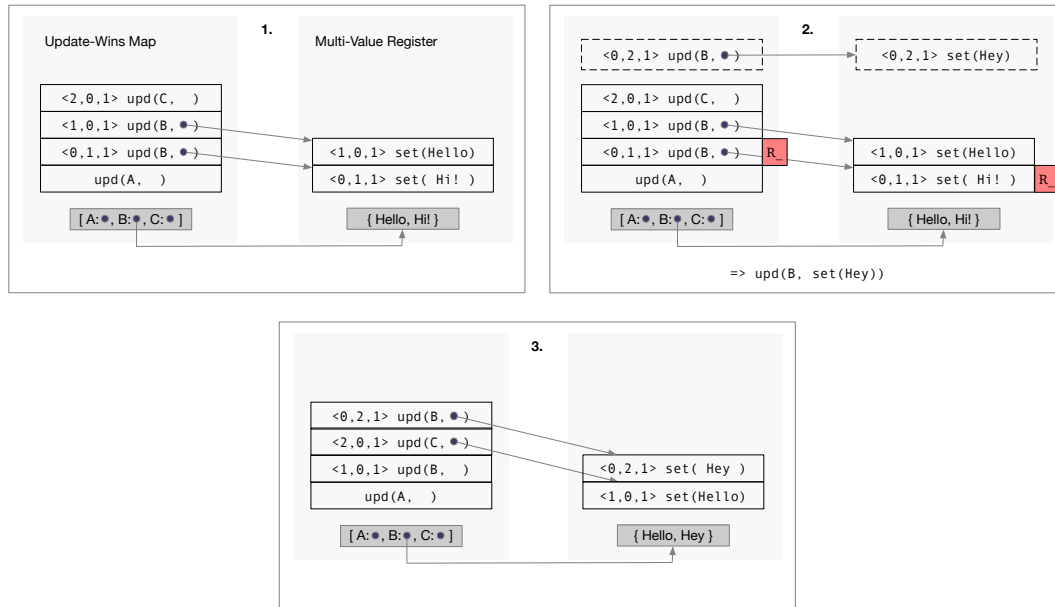
At the implementation level, a CRDT developer can decide in what manner key lookup works by providing an implementation of a particular handler function (`getChild`) that is used for lookup. The framework then provides a mechanism that allows absolute paths on a replicated structure to identify nested data structures that need to be queried or updated.

3.1.2 Updating Individual Nested CRDTs

When an operation needs to be applied to a nested child, the concurrency semantics of parent data structures must be upheld. Operations cannot just be immediately applied to the nested structure alone, as concurrent operations can be applied to the parent node which affects the key which points to the nested structure. For example, with a hash map, an entry could be concurrently updated, while it is being removed.

In our approach, when an update is applied to a particular child element, we will first issue special update operations to every parent node. These update operations signal the parent CRDTs that a nested operation is going to be applied and that it should be compared to

existing log entries using redundancy relations. For example, when building an update-wins replicated hash map, it is important to ensure that update operations win over remove operations (on the same key). At times, the update operation itself may be immediately redundant, and as such, there is no need to propagate the operation further to a nested child.



■ **Figure 3** Three stages of the internal state of a hash-map with update-wins semantics containing nested Multi-Value registers: 1) initial state, 2) arrival of an update (upd) operation, and 3) final state after applying the operation.

To illustrate how an update is applied in our approach, consider Figure 3 showing a hash map with update-wins semantics containing nested Multi-Value registers in three different stages. A Multi-Value register (MV-Register) [22] is a replicated register that, when faced with concurrent updates, will store all concurrent values. Updates that (causally) follow will replace previous values. This is in contrast to other replicated registers, for example, the Last-Writer-Wins (LWW) CRDT register [22] that always keeps a single value. When faced with concurrent updates, an LWW-Register will use an arbitrary method for picking a single update (such as picking the update from the replica with the highest network id). The first box (denoted by 1) shows the internal state and the PO-Log for the hash map and the register associated with the key 'B'. As explained, every update applied to the nested register has an associated update in the parent log. In this case, two concurrent updates were applied to the nested register, resulting in the state $\{Hello, Hi!\}$.

The second box shows the state when an `update(B, set(Hey))` is applied to the hash map. This update has a timestamp $\langle 0, 2, 1 \rangle$ which is concurrent with some operations $\langle 2, 0, 1 \rangle, \langle 1, 0, 1 \rangle$, but causally follows others $\langle 0, 1, 1 \rangle, \dots$. The update itself is applied to the hash map, making one of the existing update entries redundant, i.e., the one with vector clock $\langle 0, 1, 1 \rangle$, as it concerns the same key and has a non-concurrent timestamp. As the update operation itself is not redundant, its nested operation can be applied to the nested register. The `set(Hey)` is then applied to the nested register, making also one set operation redundant in the register, i.e., the one with vector clock $\langle 0, 1, 1 \rangle$. Note that there is another pair of concurrent operations in both the map and register that will not be made redundant, and thus are kept in the log. The third box shows the state and the log after applying `update(B, set(Hey))` resulting in the updated state $\{Hello, Hey\}$.

3.1.3 Maintaining Consistency of Children by Targeted Causal Resets

Applying redundancy checks on update operations ensures that the concurrency semantics of parents are upheld. However, they do not ensure that the concurrency semantics of children are upheld. In fact, the update mechanism ensures that redundancy relations are respected at each level of the CRDT, but these redundancy checks never cross hierarchical boundaries. This is problematic if a particular key is removed, but the remove operation is concurrent with one or more, but not all, previously applied operations (for example, remove operation c is concurrent with b , operation b is concurrent with a , but operation c causally follows operation a). This means that a key and associated child cannot be removed completely, as the child received some redundant operations (by the removal, e.g., operation a) and others that are not redundant (e.g., operation b).

To solve this issue, we introduce a novel nested redundancy relation R_n that allows nested children to be reset to a particular logical timestamp (inclusive or exclusive of concurrent operations). Using this relation, redundancy rules can be implemented that define hierarchical relations between log entries.

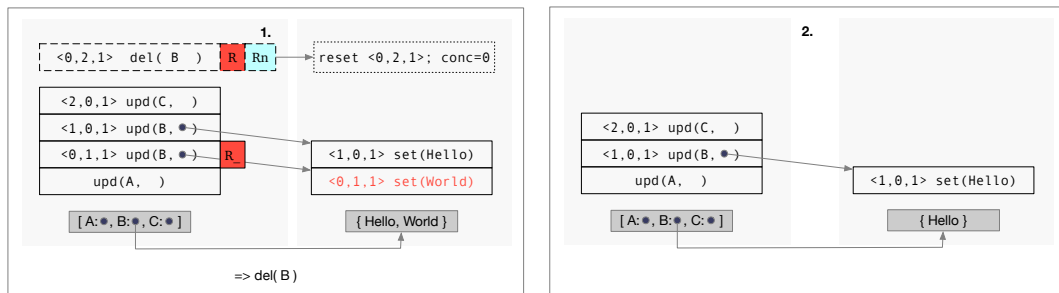


Figure 4 Example of a nested redundancy relation that selectively resets nested children, triggered by the deletion of a key. As the arriving delete (del) operation is concurrent with an update (upd) that arrived earlier, the nested child needs to be partially reset.

Figure 4 illustrates the use of the R_n relation in an update-wins hash map containing nested Multi-Value registers. The first box (denoted by 1) shows the internal state and the PO-Log for the hash map, and the register associated with the key 'B' when a $\text{delete}(B)$ operation arrives. As this operation is concurrent with one of the earlier updates in the map, and the map follows update-wins semantics, the key itself cannot be removed. The entry with a preceding vector clock $\langle 0,1,1 \rangle$, however, will be marked redundant by the existing R relation. At this point, the register associated with key B has partially redundant data, and as such needs to be updated to respect the remove operation. To this end, the R_n relation can be used to reset all operations *in the nested register* that are previous to the delete operation. In the case of the example, the set of the value 'Hi' (denoted in red in the figure) will be made redundant and removed from the register log. The second box shows the state and the log after applying the $\text{delete}(B)$ operation in which all redundant operations are removed from the entire hierarchy, and the state of the register is updated to $\{\text{Hello}\}$.

In the following section, we provide a more formal specification of our approach and extensions to the pure operation-based framework and describe example implementations for update-wins and delete-wins hash maps.

3.2 Formalised Semantics for Extended Functionality

We now describe our approach as an extension of the formal model of a pure operation-based CRDTs framework (cf. Section 2). Algorithm 2 describes the distributed algorithm for our novel nested pure operation-based framework specifying the interaction between the RCB middleware and the framework. The original Algorithm 1 used the i variable to denote a particular replica. In our extended model, Algorithm 2 compounds this with a list variable p , which denotes the path to the CRDT, relative to its parent. The top-most data structure is denoted as $root$. For example, $\{root, bob, favourite_colours\}$ could be a path that refers to a `favourite_colours` object associated with the key 'bob' in a map.

Compared to the original pure operation-based design, Algorithm 2 features new primitives for broadcasting and delivering nested operations:

- **broadcast_nested $_{i,p}(o)$** : broadcasts nested operations ensuring that the operation will be delivered to all replicas (reliably and in causal order). In our design, a broadcast can only be triggered from the top-most data structure, as such p will always be $root$.
- **deliver_nested $_{i,p}(t, o)$** : called when an operation o is delivered (e.g. after it was previously broadcasted) on a replica i at path p with causal clock t .
- **nested_operation $_i(p, o)$** : called when a nested operation o needs to be applied at path p .

Recall from Section 3.1.2 that when an operation is applied to a nested child, at each level of the parent hierarchy an `update` operation needs to be applied so that all redundancy rules can be activated. In the algorithm, the implementation of `nested_operation` ensures that an operation is packaged in an `update` operation and broadcasted using `broadcast_nested`. These broadcasted operations are received by the top-level data structure ($root$) using `deliver_nested`. `deliver_nested` will then try to deliver the operation to the child data structure specified by the path. At each level of the path, it will apply the `update` operation, check if the operation is not redundant, and if not, recursively descend into the hierarchy until the path only consists of one final child. It will then apply the actual operation to the last nested data structure using the non-nested `deliver` callback. Our approach extends the original `deliver` function with our novel nested redundancy relation: an implementation can use R_n to select what timestamps should become redundant for which nested children. Children are then (recursively) reset using the `reset` function, which takes a timestamp t and a variable $conc$ that denotes whether the reset is exclusive (only entries that happened-before) or exclusive (including all concurrent entries).

In the following section, we explore how an actual nested CRDT can be built using our proposed extensions.

3.3 Nested Pure Operation-Based Maps

In this section, we illustrate our framework by describing the design of two novel nested map CRDTs: an update-wins map (UW-Map) and a remove-wins map (RW-Map).

Table 2 shows the semantics for the update-wins map (UW-Map) in our pure operation-based framework which were informally described in the examples in Section 3.1. The design of the UW-Map CRDT is inspired by the add-wins Set CRDT [3, 4], with some modifications to take care of its nested nature [19]. The R relation for the UW-Map defines that `delete` operations will never be stored in the log (i.e., they are immediately redundant). They will, however, make any existing operation in the log redundant if they happened before (R_-). This ensures that keys can be deleted. Note that the R_- relation also makes `update` operations with the same key that happened before be redundant. This makes the data

■ **Algorithm 2** Distributed algorithm (for a replica i) showing the interaction between the RCB middleware and the pure operation-based CRDT framework.

```

state:  $s_{i,p} := \emptyset$ 
state:  $children_{i,p}$ 
on  $operation_i(o)$  :
  |  $broadcast_{i,root}(o)$ 
on  $nested\_operation_i(p, o)$  :
  |  $broadcast\_nested_{i,root}(update(p, o))$ 
on  $deliver\_nested_{i,p}(t, update((child, \emptyset), o))$  :
  |  $deliver_{i,p}(t, update(child))$ 
  |  $deliver_{n,child}(t, o)$  if  $(t, update(child)) \not\mathcal{R} s_{i,p}$ 
on  $deliver\_nested_{i,p}(t, update((child, p), o))$  if  $p \neq \emptyset$  :
  |  $deliver_{i,p}(t, update(child))$ 
  |  $deliver\_nested_{n,child}(t, update(p, o))$  if
  |  $(t, update(child)) \not\mathcal{R} s_{i,p}$ 
on  $deliver_i(t, o)$  :
  |  $s_{i,p} := (s_{i,p} \setminus \{(t', o') \mid \forall (t', o') \in s_{i,p} \cdot (t', o') \mathcal{R}_\perp (t, o)\}) \cup \{(t, o) \mid (t, o) \not\mathcal{R} s_{i,p}\}$ 
  |  $reset_{i,child}(t, 0) \mid \forall child \in children_{i,p} \cdot (child, 0) \mathcal{R}_n (t, o)$ 
  |  $reset_{i,child}(t, 1) \mid \forall child \in children_{i,p} \cdot (child, 1) \mathcal{R}_n (t, o)$ 
on  $stable_{i,p}(t)$  :
  |  $s_{i,p} := stabilize_{i,p}(t, s_{i,p})[(\perp, o)/(t, o)]$ 
  |  $stable_{i,child}(t) \mid \forall child \in children_{i,p}$ 
on  $reset_{i,p}(t, conc)$  :
  |  $s_{i,p} := s_{i,p} \setminus \{(t', o') \mid \forall (t', o') \in s_{i,p} \cdot ((t' \prec t) \vee (conc \neq 0 \wedge t' \parallel_c t))\}$ 
  |  $reset_{i,child}(t, conc) \mid \forall child \in children_{i,p}$ 

```

structure a bit more efficient. Finally, the R_n relation for UW-Map defines that all nested operations that happened before any delete need to be recursively reset (i.e. removed). As this remove should be exclusive, i.e., no concurrent entries should be removed, we additionally encode that $conc$ should be zero.

■ **Table 2** Update-wins pure operation-based map, with support for nested CRDTs.

Framework	$(t, o) \mathcal{R} s = op(o) = \text{delete}$ $(t', o') \mathcal{R}_\perp (t, o) = t' \prec t \wedge \text{arg}(o) = \text{arg}(o')$ $(child, conc) \mathcal{R}_n (t, o) = conc = 0 \wedge op(o) = \text{delete} \wedge \text{arg}(o) = child$ $stabilize(t, s) = s$
User	$update(p, o) = nested_operation([op=update, arg=[p, o]])$ $delete(c) = operation([op=delete, arg=e])$

An alternative to update-wins is ensuring that delete operations are ordered after concurrent updates, leading to a map with remove-wins semantics. Note that there are different ways to implement a CRDT from a sequential data type as there is no one solution for dealing with concurrent updates. Nevertheless, it is important to offer different variants to the end-user, as some concurrent semantics may be preferred over others in particular applications.

Table 3 shows the implementation of such a remove-wins map (RW-Map) in our framework. It is structured similarly to the AW-Map but has some additional complexity as the log needs to retain all delete operations until they are causally stable. The R_n relation encodes that all previous or concurrent nested updates need to be removed (to ensure remove-wins semantics).

■ **Table 3** Remove-wins pure operation-based map, with support for nested CRDTs.

Framework	$(t, o) \mathbb{R}_s$	$= \text{op}(o) = \text{update}$ $\wedge (\exists (t', o') \in s \cdot \text{arg}(o) = \text{arg}(o') \wedge \text{op}(o') = \text{delete} \wedge t \parallel_c t')$
	$(t', o') \mathbb{R}_\perp (t, o)$	$= t' \prec t \wedge \text{arg}(o) = \text{arg}(o') \wedge \text{op}(o) = \text{delete}$
	$(\text{child}, \text{conc}) \mathbb{R}_n (t, o)$	$= \text{op}(o) = \text{delete} \wedge \text{arg}(o) = \text{child}$
	$\text{stabilize}(t, s)$	$= s$
User	$\text{update}(p, o)$	$= \text{nested_operation}([\text{op}=\text{update}, \text{arg}=[p, o]])$
	$\text{delete}(c)$	$= \text{operation}([\text{op}=\text{delete}, \text{arg}=c])$

In this design of an RW-Map, in theory, `update` operations do not need to be stored in the log as these updates are stored in the nested children. However, only the last update operation for a particular child is kept (since previous update operations are removed from the log as they are redundant) As such, storing the `update` operations in the log can be useful to check if a particular child has a value, without having to query the nested children. When storing these entries poses a problem memory-wise, they can trivially be removed with no impact on the behaviour of the data type.

The implementation of these map CRDTs demonstrates that supporting nested structures can be tackled in a structured and easy way. Our framework handles all logic related to nesting and update propagation, aiming to provide an easy-to-use interface. Additionally, hierarchical redundancy rules can be encoded using the \mathbb{R}_n relation, ensuring that concurrency semantics are upheld at any level.

3.4 Discussion

We believe that our approach simplifies the design of replicated nested CRDTs, and with it, we aim to reduce their implementation complexity. With the presented methodology, one can think of every CRDT with nesting support as a flat CRDT, which needs to support one additional operation, namely `update`. For example, a map is similar to a set of keys with an associated value. In a set, we can add and remove keys. Using some rules we can make the set add-wins or remove-wins, and with a bit of extra work, we can define how an `update` operation could be ordered against concurrent add and remove. This could be the core design of a Map. Our framework will make sure that every nested operation, e.g. a nested operation to a child of the map, is first represented as an `update` operation for the parent CRDT. The parent CRDT (e.g. the map) does not need to know anything about the nested content of this update, it is simply trying to make sure that this update will be properly ordered between the additions and removals of keys. This alone, however, is not enough to ensure convergence, i.e. that the algorithm is functional and correct. Depending on the arrival order of an update in combination with other concurrent operations, the associated nested operation may have been applied to some replicas and not to others. To ensure that the nested state converges, the algorithm sometimes might need to apply some cleanup procedures, which is precisely where the nested redundancy relation comes into play. In Section 5.1 we formally prove that this is the case for our approach and our implemented designs.

4 Implementation

We implemented our novel nested pure operation-based approach in Flec [5, 6], an extensible programming framework and middleware for CRDTs written in TypeScript. Flec incorporates the concepts of ambient-oriented programming [10, 12], to discover and communicate with

replicas in a distributed dynamic network. Since it has support for pure operation-based CRDTs and RCB for causal delivery, Flec is the ideal platform for implementing our approach. In this section, we describe the extensions and modifications to Flec that are required to support nested pure operation-based CRDTs.

4.1 Nesting in Flec

To support the implementation of pure operation-based CRDTs, Flec provides an open framework with the following operations:

- **isPrecedingOperationRedundant** and **isConcurrentOperationRedundant**: encode the $\mathbf{R}__$ (or R_0, R_1) binary relation(s) defining if existing log entries become redundant by a new operation. Alternatively, **isRedundantByOperation** unifies both methods.
- **isArrivingOperationRedundant**: Encodes the \mathbf{R} binary relation (i.e., is a new operation redundant by an already existing log entry).
- **onLogEntryStable**: performs an action when an operation becomes stable.
- **onRemoveLogEntry**: performs an action when a particular item is removed from the log (for example if it was marked redundant by **isRedundantByOperation**).
- **onAddLogEntry**: performs an action when a new operation arrives in the log.

To build an actual CRDT data type, developers have to implement these methods, following the semantics of the datatype. While **onLogEntryStable**, **onRemoveLogEntry**, and **onAddLogEntry** are not required to implement the CRDT semantics, they can help optimise a pure operation-based CRDT to use a native data structure for causally stable entries. The log, entries, and optional native data compacted structures can be queried using the following methods:

- **getLog**: gets all current log entries.
- **getState**: gets all current log entries, the compact native state, and the current logical timestamp for the replica.
- **getConcurrentEntries**: gets all concurrent log entries for an operation.

In this work, we extend the framework with the following new hooks and operations to implement nested pure operation-based designs:

- **setChildInitialiser**: is a method that will be used to initialise new children, using child-specific constructs (e.g. if you want children to be AW-Sets, the initialiser will return a new AW-Set).
- **doesChildNeedReset**: encodes the \mathbf{R}_n binary relation (i.e., from what timestamps do children need a partial reset).
- **performNestedOp**: performs a nested operation and broadcasts it to other replicas.
- **addChild**: register a CRDT as a child to a parent, for a particular key.
- **resolveChild**: override the default internal child bookkeeping and instruct the framework on how to resolve a particular child CRDT based on a name (this will disable **addChild**).

4.2 Implementing Nested CRDTs in Flec

We now illustrate the extended Flec by means of the RW-Map CRDT described in Table 3. Listing 1 and Listing 2 show the core of the implementation of RW-Map CRDT in Flec. Lines 4 to 8 in Listing 1 define the CRDT constructor, which is used to initialise the **values** property that contains all nested children. Additionally, an initialiser can be specified that sets the initial (start) value for children. For example, if a map with a nested AW-Set is needed, the initializer will initialize a new AW-Set CRDT. Lines 14–16 in Listing 1 show the

update function which can be used to apply nested operations on children (by CRDT client code). Any operation on a child is indicated by specifying a particular path, and the update to be applied. Using `performNestedOp` this operation will be propagated to the child and all replicas. The actual semantics can be seen in Listing 2 which shows the implementation of the redundancy relations and children referencing.

■ **Listing 1** The implementation of an RW-Map in Flec, using the described extensions (A).

```

1  export class RRWMap extends PureOpCRDT<MapOps> {
2    values: Map<string, NestedCRDT>;
3
4    constructor(initializer: () => NestedCRDT) {
5      super();
6      this.values = new Map();
7
8      this.setChildInitialiser(initializer);
9    }
10   ...
11   // User functions
12   ...
13
14   public update(path, ...args) {
15     this.performNestedOp("update", path, args);
16   }
17 }

```

Lines 20 to 22 in Listing 2 show the implementation of the `resolveChild` method which allows the underlying Flec framework to reference children, stored in the `values` property. The rest of the listing shows how the RW-Map implements redundancy relations to achieve remove-wins semantics: the RW-Map provides an implementation for `isPrecedingOperationRedundant` to implement the R_{-} relation: any operation in the log is redundant if it has happened before a newly arriving operation, and if they are acting upon the same child. It also implements `isArrivingOperationRedundant` to define the R relation: any arriving update is not applied if a concurrent delete is stored in the log. Finally, by providing an implementation for `doesChildNeedReset` we specify that when a delete arrives for a particular child, the child will be reset. The `reset_concurrent` flag is set to true to indicate that even concurrent updates to the child should become redundant.

■ **Listing 2** The implementation of an RW-Map in Flec, using the described extensions (B).

```

1  protected isPrecedingOperationRedundant(existing: MapEntry, arriving
2    : MapEntry, isRedundant: boolean) {
3    return arriving.isDelete() && existing.hasSameArgAs(arriving);
4  }
5
6  protected isArrivingOperationRedundant(arriving: MapEntry) {
7    const concurrentDeletes = this.getConcurrentEntries(arriving).
8      filter(e => e.entry.isDelete() && e.entry.hasSameArgAs(
9        arriving));
10   return concurrentDeletes.length > 1;
11 }
12
13 protected doesChildNeedReset(child, arriving: MapEntry) {
14   return {
15     condition      : arriving.isDelete() && arriving.args[0] ==
16       child,
17     reset_concurrent: true
18   };
19 }
20
21 // Resolve child CRDTs
22 protected resolveChild(name: string) {
23   return this.values.get(name);
24 }

```

5 Validation

To validate our work, we conduct three experiments. First, verify the correctness of our proposed framework and nested pure op-based maps. Secondly, we implement the concepts in a real programming framework and finally, we compare it to another framework featuring similar concepts.

5.1 Verification with VeriFx

In order to verify our approach, we have re-implemented the core of our nested pure operation-based CRDTs in VeriFx [11]. VeriFx is a programming language for replicated data types with automated proof capabilities that allow users to implement replicated data types in a high-level language and express correctness properties that are verified automatically. VeriFx internally uses an SMT theorem prover to search for counterexamples for each property that needs to be upheld. It also enables the transpilation of the data types to mainstream languages (e.g. Scala and JavaScript).

Correctness means that strong eventually consistent data types can be built with the framework and that they exhibit the *strong convergence* property which requires that replicas need to have received the same operations to be in the same state (regardless of the order in which the operations have been received). Shapiro et al. showed in [22] that operation-based CRDTs guarantee strong convergence if all concurrent operations commute. In our case, this implies checking the effects of all redundancy relations. Proving the correctness is, however, slightly trickier in our case, as we are dealing with a recursive design. SMT solvers, such as Z3 used by VeriFx, do not deal well with recursive and nested data structures, as they might not be able to find a solution in a finite time. To verify our approach, we thus combine VeriFx proofs with structural induction, which limits the recursion depth needed to verify our design:

- Base case: we implemented a 'perfect' resettable pure operation-based CRDT in VeriFx that can model both a flat CRDT or a CRDT containing children. The CRDT logs all operations in a single flattened log (e.g., one log for all potentially nested structures). Items in the log can be reset by a parent when requested. No redundancy rules are applied. This design ensures that we can represent a 'correct' nested structure (in terms of SMT assumptions) without needing a recursive model. We use a VeriFx proof to ensure convergence of this 'perfect' CRDT.
- Induction step: a particular nested CRDT can be implemented on top of our VeriFx implementation and set to use perfect nestable CRDTs as children. With this approach, VeriFx can then be used to prove that our approach is correct for one level of nesting, for all pairs of operations.

By combining the base case and induction step, we prove using structural induction that our framework remains correct for any nestable structure.

■ Listing 3 Convergence update-update.

```

1  proof FUWMap_update_update_converges {
2    forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
      : VersionVector, o1: SimpleOp, o2: SimpleOp) {
3      ( t1.concurrent(t2)  && map.children.contains(k1) && map.
        children.contains(k2) &&
4          map.polog.forall((e: TaggedOp [FMapOp])=>
            ((e.t.before(t1) || e.t.concurrent(t1)
              )))
5          && (e.t.before(t2) || e.t.concurrent(t2)
              ))) => (

```



```

6
7     map.update(t1, k1, o1).update(t2, k2, o2)
8     ==
9     map.update(t2, k2, o2).update(t1, k1, o1)
10    )
11  }
12  }

```

■ **Listing 4** Convergence update-delete.

```

1  proof FUWMap_update_delete_converges {
2    forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
3      : VersionVector, o1: SimpleOp) {
4      (t1.concurrent(t2) && map.children.contains(k1) &&
5        map.polog.forall((e:TaggedOp[FMapOp])=>((e.t.before(t1) || e.
6          t.concurrent(t1)) && (e.t.before(t2) || e.t.concurrent(t2)
7            )))) =>: (
8        map.update(t1, k1, o1).delete(t2, k2)
9        ==
10       map.delete(t2, k2).update(t1, k1, o1)
11     )
12   }
13 }

```

■ **Listing 5** Convergence delete-delete.

```

1  proof FUWMap_delete_delete_converges {
2    forall(map: FUWMap, k1:String, k2: String, t1: VersionVector, t2
3      : VersionVector, o1: SimpleOp, o2: SimpleOp) {
4      (t1.concurrent(t2) && map.children.contains(k1) && map.children.
5        contains(k2) && map.polog.forall((e:TaggedOp[FMapOp])=>((e.t.
6          before(t1) || e.t.concurrent(t1))
7            && (e.t.before(t2) || e.t.concurrent(t2))
8              ))) =>: {
9      map.delete(t1, k1).delete(t2, k2) == map.delete(t2, k2).delete
10     (t1, k1)
11   }
12 }

```

As an example, Listings 3, 4, and 5 show the VeriF_x proof logic that was used to check the behaviour of concurrent operations on an update-wins map implemented with our framework. We define that any pair of correct operations that are concurrent and applied to a correct state should commute. The operations and state are correct if the operations (causally) follow or are concurrent with all other operations that were applied previously to the state (e.g. everything in the log). For this definition, we assume the usage of RCB (which is the case with the pure operation-based CRDT framework), so that we know that everything in the log must be concurrent or happened-before. In other words, the logic encodes the correctness properties that should always hold in our framework, i.e. that if all operations on the map commute and the nested operations are applied to correct CRDTs (in our case, all nested operations are applied to a 'perfect' CRDT), that the map is correct.

We use the automatic VeriF_x prover to verify these properties hold given the implemented designs. Internally, the VeriF_x SMT engine will look for valid solutions that satisfy the negation of our definitions, it will search for any case where the correctness properties are violated. Since no counterexamples (valid solutions for the negation of properties) were found after exhausting all search options, we can then constitute that our framework model is valid according to the correctness properties.

■ **Table 4** Implemented nested CRDT types.

CRDT	Semantics
UW-Map	Update-wins map where values can be CRDTs. Update win from concurrent deletes.
RW-Map	Remove-wins map where values can be CRDTs. Deletes win from concurrent updates.
RW-Map (mod)	Modular version of the remove-wins map that allows more efficient memory usage.
AW-Map	A variant of the update-wins Map where keys are managed by an add-wins set.
AW-Set	An add-wins set where values can be CRDTs.
DW-List	A delete-wins linked list where elements can be CRDTs.
ImmutableCRDT	A map with immutable keys, which behaves similarly to structs in C.

Using this approach, we have verified our map designs, validating both the concurrency semantics of our proposed CRDTs and proving that our novel framework functions correctly. The benefit of our verification approach is that to validate the correctness of any nestable CRDT (built on our framework), one only needs to encode proofs for the operations on a flat level. All needed nesting aspects of the proof will automatically be inherited from our VeriFx implementation. The full source code for our VeriFx implementation, including proofs and implemented models, is included as an artifact.

5.2 Portfolio of Nested CRDTs in Flec

To show the flexibility and applicability of our approach, we have implemented several commonly used data structures as novel nested pure operation-based CRDTs in Flec, summarised in Table 4. As shown in the previous section, we have map implementations with update-wins and remove-wins semantics. Maps form the basis for many other data structures and thus are essential to any replication framework. They have been verified using their VeriFx-based implementations and have been used in more complex data structures since.

We have implemented two other maps: one modified map (based on the remove-wins map) that optimises some structures to have better memory resource usage, and another map where keys are managed by an add-wins set. Finally, we have a delete-wins list that can be used to store values in sequential order. Similarly to other sequential replicated structures such as RGAs [13], a linked list is used internally.

The source code for the update-wins map, remove-wins map and delete-wins list implementations can be found as part of the included artifact.

5.3 Use-Case: A Mixed CRDT-Based Distributed Filesystem

To validate our approach in a real-world application scenario, we implemented a distributed file system based on the work of [25] in our Flec implementation. This application is also used later in Section 5.4 to compare our approach to state-of-art.

Flec does not only support pure operation-based CRDTs, it has many general-purpose constructs for building any replicated data type. As such, it comes with a portfolio of (non-pure-op) general CRDTs. While our extensions to Flec were focused on pure operation-based CRDTs, part of the nesting support we added can also be used in conjunction with general non-pure operation-based CRDTs to develop real-world applications.

When composing (traditional) CRDTs, operations on a (parent) root node typically trigger several operations that will be applied to internal (nested) CRDTs. For a single operation, these sub-operations need to be applied atomically, they cannot be viewed as independent and should not automatically replicate to nested children of replicated CRDTs. This is in contrast with our main approach where an update is applied via a particular sub-path. To ensure compatibility with this approach in the framework, nested children can detect the context in which operations are applied. If a nested CRDT has a parent, and an operation is applied directly from that parent (and not via a nested update), the operation will not be broadcasted to other replicas. Instead, it is assumed that the (top-)parent operation will be broadcasted, resulting in the same nested update path on other replicas.

We now discuss the overall data structures and operations of the distributed file system. Listings 6–8 in the appendix show the core of the implementation. It has been modified to hide some minor boilerplate code, type definitions, and a lot of operation handling code, but it contains the essentials. Listing 6 shows the main body of the `DistributedFS` class, which implements the core functionality of the CRDT. By extending the `SimpleCRDT` class it automatically inherits all the distribution and CRDT functionality from Flec (along with our extensions). Lines 5-21 define the required data structures for the distributed file system that keep track of metadata for files, groups and users. To this end, we define three maps, and each map on its own contains records (in the form of `ImmutableCRDT`) containing other CRDTs for storing the metadata of particular files, groups and users. For example, the `files` data structure is defined using an RW-Map and contains filesystem meta-data related to access rights, ownership, and data content. The data types we use for the registers (`AccessRightF`, `UserID`, ...) are basic types constructed from primitive types such as numbers or strings and can be stored directly in the registers. `AccessRightF` is a numerical value that we index as a bit-vector to store our permission flags (similar to POSIX systems). We provide an additional TypeScript class, `AccessRight`, that provides a high-level abstraction to this bit-vector, but concretely we store numerical values in the CRDT register. Lines 24-28 define the `onLoaded` method which associates the aforementioned three maps with their parent CRDT. In line 30, the `setHandler` method defines all operation handlers which implement the semantics of the CRDT.

Listing 7 shows the implementation of the `CreateFile` operation in more detail. Listing 8 shows code that exposes some of the CRDT API to the local user, for performing some basic actions which are used by the `test` method in Listing 9 to show local usage of the file system functionality. Flec will ensure that all operations are properly replicated and distributed. In general, most of the code is similar to that of sequential data structures, and the API is not much more complex. This is in line with the goal of our framework: an easy-to-use interface for building CRDTs where developers can immediately benefit from a middleware that does all the heavy lifting.

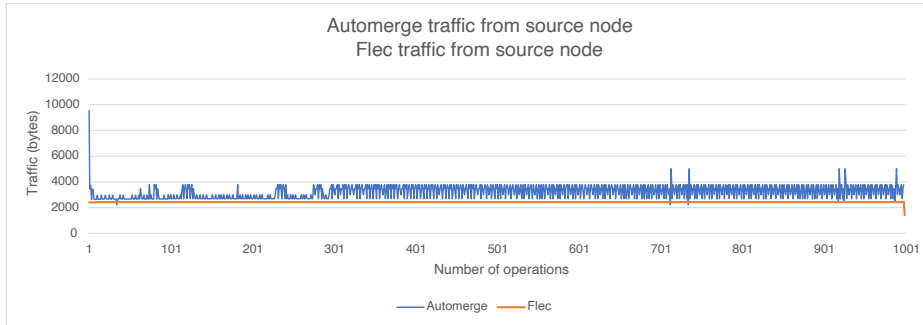
5.4 Evaluation of Network Traffic in Comparison With Automerge

To compare our approach with state of the art, we implemented the same distributed filesystem in Automerge v1.0.1 [15] and evaluated the differences in network traffic between our Flec implementation and the Automerge implementation.

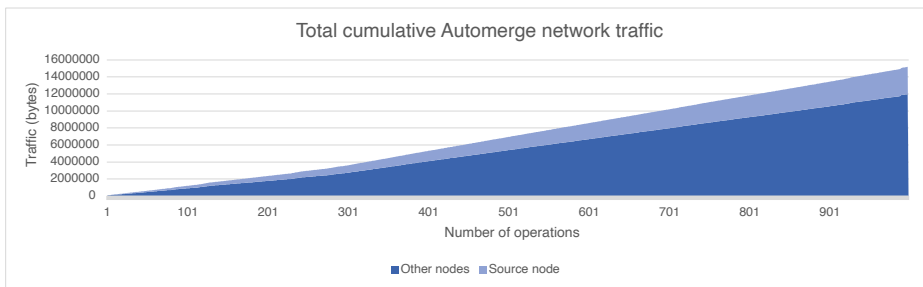
It is not possible to select the individual concurrency semantics for nested objects with Automerge, as is possible with our extension to Flec. As such, the implementation has a slight difference in concurrency semantics when compared to the original design [25] and our implementation. For example, while the distributed filesystem (DFS) specification describes update-wins concurrency semantics for the user list, the Automerge implementation uses remove-wins concurrency semantics. Functionality-wise, it has the same features. In fact, in our implementations, both the Automerge and Flec versions have the same API.

2:18 Nested Pure Operation-Based CRDTs

Automerge itself does not provide a network layer but instead provides an API that allows you to query (Automerge) documents for changes, and if any changes exist, you can propagate these over any networking channel that your application depends on. On the receiving end, you can insert these changes back into Automerge, which can merge the received information in the local state. Automerge itself uses a state-based approach, where only the required changes (deltas) are propagated instead of the full state, to conserve network bandwidth.



■ **Figure 5** Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. In every operation, a file is created and written.



■ **Figure 6** Total cumulative networking traffic (in bytes/op) from all nodes for Automerge. In every operation, a file is created and written.

For the experiments, we used a virtual network for both Automerge and Flec, which allows us to reproduce benchmarks and results with little non-determinism. We set up a system with 5 nodes (ad-hoc, peer-to-peer), and issue a thousand operations per experiment.

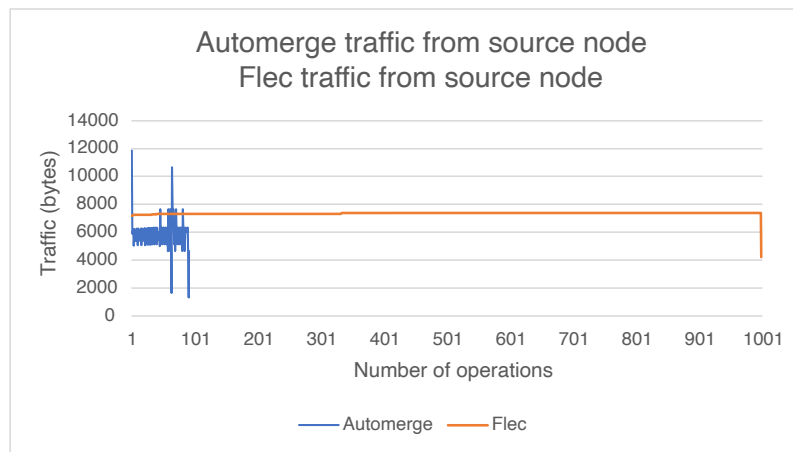
5.4.1 Experiment A: File Creation and Writing

For the first benchmark, each operation exists out of file creation and file modification. We applied these operations a thousand times to a deployed distributed file system, once using the Flec implementation and once with the Automerge implementation.

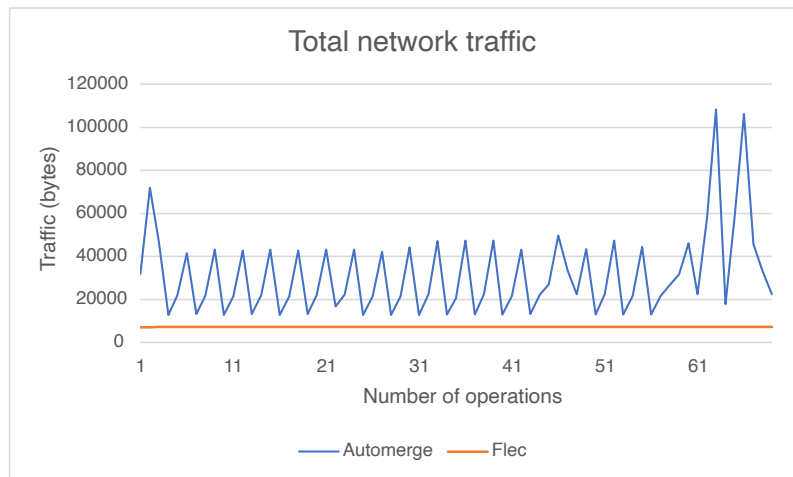
Figure 5 shows the network traffic originating from the source node (the node where the operations are applied), for both implementations. As both our approach and Automerge share the essential updates, the results are fairly stable and linear. Automerge will always send small updates containing the state delta (which means the newly modified file) and our extension to Flec sends the operations itself. While Automerge uses a binary representation for the update payload, the payload itself is still heavier than the non-optimized JSON payload used in Flec.

The visualisation hides some essential information, however. Automerge uses an additional protocol that allows replicas to propagate updates among each other. This means that not only the source node will share information, but also other nodes that received the new updates if they believe that other replicas may be missing information. Figure 6 highlights the additional traffic, showing that it makes up a significant portion of the total network traffic. In Flec updates are only sent directly from a source node to a destination node, and as such, there is no additional network usage.

5.4.2 Experiment B: User, Group, and File Creation, and Configuration

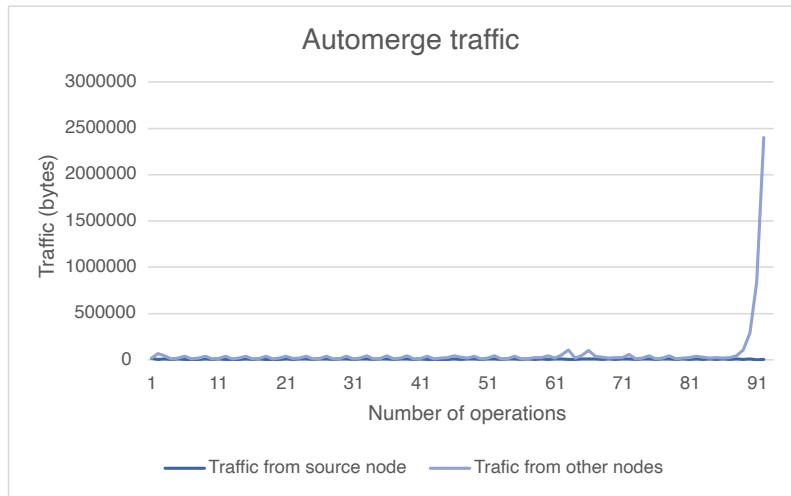


■ **Figure 7** Network traffic (in bytes/op) originating from the source node for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written.



■ **Figure 8** Total network traffic (in bytes/op) for both Automerge and Flec. Every operation creates a new user, a new group, and a new file. The user is added to the group, and the file is created with the new user as the owner. Finally, the file is written.

For the second experiment, in each operation, we create a new user, and a new user group, add the user to the new group, create a new file (with the new user as owner), and write to this file. This extra complexity leads to some interesting results. As seen in Figure 7 the



■ **Figure 9** Total network traffic for Automerge for the previous experiment, highlighting an issue with exponential growth after a certain number of operations.

Automerge measurements stop at around ~ 100 operations. This is because the additional gossip traffic starts growing exponentially (see Figure 9) and causes the entire system to halt. We are not exactly certain what causes this problem, but we did not observe this issue with the previous experiment, only when we applied more complex operations. We believe that this is not correct behaviour from Automerge but have not been able to identify the root cause of the bug yet. The behaviour is consistent and reappears with each run. To be able to evaluate this example anyway, we will only focus on the initial measurements before the exponential explosion. Based on Figure 7 we can see that Automerge has a lower network overhead on the source node when compared to Flec. When looking at the total traffic, however (Figure 8), we can see that Automerge still utilizes more bandwidth. The reason for this is that as we are sending many operations, other replicas start propagating updates as well, resulting in the source node itself sending fewer updates (as it is relieved from work).

5.4.3 Experimental Evaluation: Conclusion

With this experimental evaluation, we showed that our approach is comparable to state-of-the-art CRDT frameworks, even though Flec and our extensions have not yet been optimised for non-experimental use. While additional optimisations can be applied to the pure operation-based CRDT framework and our nested framework extension, these results are promising and show that our approach is viable in real-world scenarios.

We now discuss some of the potential threats to the validity of our experimental evaluation and why our benchmark methodology and conclusions are not invalidated by these threats.

- T: The number of replicas used in our benchmarks (5) is potentially too low.
- The results of the experiments show that this number is fair, as it allows us to observe interesting differences between both benchmarked platforms. For example, in Figure 6., we can see that the total traffic generated by Automerge in experiment A quickly exceeds the traffic of our approach, but we can still compare results in a reasonable way.
- T: The chosen experiments are not realistic.
- The operations are tailored to induce complicated internal behaviour of the replicated data type, which we expect to also occur doing normal and realistic tasks. Of course, in a realistic setting such operations may not be applied repeatedly, but in the context

of our evaluation we wanted to evaluate behaviour under repeated, continual usage while testing many different parts of the CRDT framework as well. However, the total amount of operations used in the benchmarks could be achieved over a small period in a real deployment, and therefore it is important that a distributed filesystem system can handle such load. The operations used aim to use nesting to its full extent, in a realistic application case (a distributed file system). We, therefore, believe that the benchmarks are suitable for evaluating our approach.

- T: The benchmarks only compare results with one other related work.
- While comparing with extra platforms could improve the evaluation, we do not believe that this invalidates or diminishes our results. Automerge is a state-of-the-art framework for replicated data structures, with a lot of usages, and therefore a proper framework to compare against and evaluate whether our proposed approach has viability.

6 Related Work

The bulk of research in replicated data types has focused on devising a portfolio of conflict-free data structures such as counters, sets, and linked lists [22, 24, 20, 7, 21, 19]. However, the composition and nesting of CRDT have drawn little attention so far. The composition of replicated structures is possible in a few frameworks like Automerge [15] and Lasp [17]. While Automerge allows programmers to arbitrarily nest linked lists and maps in a document, it doesn't allow for much flexibility regarding the actual merging semantics. Lasp supports functional transformations over existing CRDTs provided in the language, which allows a composition to some extent. However, when the current portfolio of CRDTs falls short in those frameworks, developers need to design the desired nested data structure from scratch. This requires rethinking the data structure completely such that all operations commute and manually implement conflict resolution for concurrent non-commutative operations, which is hard and error-prone [22, 15, 1].

Weidner et al. [23] explore ways to compose and de-compose pure operation-based CRDTs. They introduce techniques for creating novel CRDTs based on existing (de-composed) CRDTs *with a static structure*. They do not aim to provide a solution for creating general nested data structures, but instead, propose constructs to define the semi-direct product of op-based CRDTs. This means that instead of nesting and maintaining individual semantics, novel semantics are introduced to create a combination of several CRDTs, leading to an entirely new, non-nested CRDT. In our approach, nested data structures can change dynamically during runtime, using maps, lists, and sets.

Preguiça in [19] explains several possible nesting semantics for operation-based CRDTs. To support a wide variety of CRDTs as nested values in different settings, it will be necessary for the CRDTs to be able to partially reset themselves to an initial state before a particular timestamp. Typically, this means that this reset has to be recursive and that nested sub-CRDTs will need to be reset as well. Without a disciplined approach, combining ad-hoc CRDTs will be hard. The benefit of using a log-based approach, which we are proposing, is that such recursive resets can be supported at the framework level, in a unified way, without needing to modify the semantics of CRDTs.

Operation-based and state-based CRDTs are two approaches to guarantee SEC that share an equivalence to some extent. While both approaches can be emulated as each other [22], it depends on the application or system in use which approach might be more suitable. It is typically a tradeoff choice, between waiting for the right moment to make a state merge, or rather propagating operations continuously. It should be possible to emulate our approach

(and pure operation-based CRDTs in general) as a state-based design, but making it efficient might be problematic as one would need to keep track of extra meta-data related to the applied operations (in order to maintain individual semantics between nested components). This information comes for free in an operation-based CRDT approach; as the operations themselves are directly propagated.

7 Conclusion

Conflict-Free Replicated Data Types (CRDTs) are useful programming tools to replicate data in a distributed system as they guarantee that eventually, all replicas end up in the same state. In this paper, we explore a structured approach for designing nested CRDTs based on the ideas of pure operation-based CRDTs. We propose a novel framework for building nested pure operation-based CRDTs and show how several common nested data structures can be designed and modelled in the framework. We validate our approach by extending an existing pure operation-based framework written in TypeScript, Flec, to include support for nested pure operation-based CRDTs and implement a portfolio of commonly nested data structures. This portfolio includes novel add-wins and remove-wins pure operation-based CRDTs, implemented following our framework. Additionally, we demonstrate the flexibility of the framework by implementing a distributed filesystem model using these techniques. We used an SMT-based implementation to verify the correctness of our approach. Finally, showed that our approach produces competitive results compared to Automerge, a state-of-the-art framework.

References

- 1 P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. *CoRR*, abs/1410.2803, 2014. [arXiv:1410.2803](#).
- 2 C. Baquero, P. S. Almeida, and A. Shoker. Making operation-based crdts operation-based. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, pages 126–140, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- 3 C. Baquero, P. S. Almeida, and A. Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017. [arXiv:1710.04469](#).
- 4 J. Bauwens and E. Gonzalez Boix. Improving the reactivity of pure operation-based crdts. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447865.3457968.
- 5 J. Bauwens and E. Gonzalez Boix. Flec: A versatile programming framework for eventually consistent systems. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3380787.3393685.
- 6 J. Bauwens and E. Gonzalez Boix. From causality to stability: Understanding and reducing meta-data in crdts. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, MPLR '20, pages 3–14, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3426182.3426183.
- 7 Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balesgas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint*, 2012. [arXiv:1210.3368](#).
- 8 K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987. doi:10.1145/7351.7478.

- 9 S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP'12*, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_14.
- 10 T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix., J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Society of Computer Science (SCCC'07)*, pages 3–12, Iquique, Chile, 2007. doi:10.1109/SCCC.2007.12.
- 11 Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. Verifx: Correct replicated data types for the masses, 2022. doi:10.48550/ARXIV.2207.02502.
- 12 J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 230–254, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 13 R. Hyun-Gul, J. Myeongjae, K. Jin-Soo, and L. Joonwon. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.
- 14 G. Kaki, S. Priya, KC Sivaramakrishnan, and S. Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360580.
- 15 M. Kleppmann and A. R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel & Distributed Systems*, 28(10):2733–2746, October 2017. doi:10.1109/TPDS.2017.2697382.
- 16 R. Klopheus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming, CUFPP '10*, pages 14:1–14:1, New York, NY, USA, 2010. ACM. doi:10.1145/1900160.1900176.
- 17 Christopher Meiklejohn and Peter Van Roy. Lasp: A Language for Distributed, Coordination-free Programming. In *17th Int. Symp. on Principles and Practice of Declarative Programming, PPDP '15*, pages 184–195, 2015.
- 18 Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 2016 ACM International Conference on Supporting Group Work, GROUP '16*, pages 39–49, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2957276.2957310.
- 19 N. Preguiça. Conflict-free replicated data types: An overview, 2018. arXiv:1806.10254.
- 20 Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.
- 21 M. Shapiro. Replicated Data Types. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia Of Database Systems*, volume Replicated Data Types, pages 1–5. Springer-Verlag, July 2017. doi:10.1007/978-1-4899-7993-3_80813-1.
- 22 M. Shapiro, N Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.
- 23 Matthew Weidner, Heather Miller, and Christopher Meiklejohn. Composing and decomposing op-based crdts with semidirect products. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3408976.
- 24 Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Trans. on Parallel and Distributed Systems*, 21(8):1162–1174, August 2010.
- 25 Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. Access control conflict resolution in distributed file systems using crdts. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC '21*, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447865.3457970.

A DFS Code Listings

This appendix contains code listings with portions from our distributed filesystem test implementation. A legend for the used types can be found in Table 5.

■ **Table 5** Legend for the TypeScript classes and types used in the DFS implementation.

Class / Type	Description
RWWMap	Nested Remove-Wins Map CRDT.
RUWMap	Nested Update-Wins Map CRDT.
ImmutableCRDT	ImmutableCRDT map. Nested CRDT map that works as a C struct.
Register<T>	LLW-Register CRDT, containing a primitive value of type T.
AccessRightF	Alias of the 'Number' type, represents a bit vector with access flags.
AccessRight	Abstraction over AccessRightF, never stores in a CRDT, just used for easy modification of the access right bit vectors.
SimpleCRDT	Abstract CRDT class in Flec, for creating operation-based CRDTs.
GroupID / UserID / FileID	Aliases for strings that represent UUIDs.

■ **Listing 6** The general structure of the DFS nested CRDT, highlighting the main nested children that contain the filesystem meta-data.

```

1 export class DistributedFS extends SimpleCRDT<FSOperation> {
2   handler: FSOperation;
3   ...
4
5   files = new RRWMap(t => new ImmutableCRDT({
6     access_right_owner: new Register<AccessRightF>(),
7     access_right_group: new Register<AccessRightF>(),
8     access_right_other: new Register<AccessRightF>(),
9     file_owner: new Register<UserID>(),
10    file_group: new Register<GroupID>(),
11    file_data: new Register<string>()
12  }));
13
14  groups = new RRWMap(t => new ImmutableCRDT({
15    group_users: new AWSet(), // must be RW
16    created: new Register<flag>()
17  }));
18
19  users = new RUWMap(t => new ImmutableCRDT({
20    is_admin: new Register<flag>()
21  }));
22  ...
23
24  onLoaded() {
25    this.addChild("files", this.files);
26    this.addChild("users", this.users);
27    this.addChild("groups", this.groups);
28  }
29
30  setHandler() {
31    const me = this;
32    this.handler = {
33
34      ChangeOwner(userId: UserID, newOwnerId: UserID, fileId: NodeID
        ) { ... },

```

```

35     ChangeGroup(userId: UserID, newGroupId: GroupID, fileId:
        NodeID) { ... },
36     ChangeOwnerPermission(userId: UserID, newPerm: AR, fileId:
        NodeID) { ... },
37     ChangeGroupPermission(userId: UserID, newPerm: AR, fileId:
        NodeID) { ... },
38     ChangeOtherPermission(userId: UserID, newPerm: AR, fileId:
        NodeID) { ... },
39     ...
40     CreateUser(with_admin_rights: boolean, id: string) { /* ... */
        },
41     CreateGroup() { /* ... */ },
42     AssignUserToGroup(authorId: UserID, groupId: GroupID, userId:
        UserID) { ... },
43     CreateFile(userId: UserID, groupId: GroupID, fileId: NodeID) {
        ... see listing below ... },
44     WriteFile(userId: UserID, fileId: NodeID) { ... },
45     ...
46     update(key: string) { }
47 }
48 }
49 }

```

■ **Listing 7** Structure of the operation handling code for the DFS. Included is the code for the CreateFile callback, which can either be invoked locally or as a result of a replicated operation.

```

1  setHandler() {
2      const me = this;
3
4      this.handler = {
5          ...
6
7          CreateFile(userId: UserID, groupId: GroupID, fileId: NodeID) {
8              const user = me.users.lookup(userId) as any;
9              const group = me.groups.lookup(groupId) as any;
10
11             if (group && user && group.group_users.contains(userId)) {
12                 console.log("adding file");
13
14                 me.files.update([{ key: fileId, op: "update" },
15                                 { key: "file_owner", op: "write" }], userId);
16                 me.files.update([{ key: fileId, op: "update" },
17                                 { key: "file_group", op: "write" }], groupId);
18
19                 const isAdmin = user.is_admin.is(FLAGS_TRUE);
20                 const access_owner = new AccessRight(isAdmin, true, true);
21                 const access_group = new AccessRight(isAdmin, true, false);
22                 const access_other = new AccessRight(isAdmin, true, false);
23
24                 this.files.update([{ key: fileId, op: "update" },
25                                 { key: "access_right_owner", op: "write" }], access_owner.
26                                     toEnum());
27                 this.files.update([{ key: fileId, op: "update" },
28                                 { key: "access_right_group", op: "write" }], access_group.
29                                     toEnum());
30                 this.files.update([{ key: fileId, op: "update" },
31                                 { key: "access_right_other", op: "write" }], access_other.
32                                     toEnum());
33             }
34         },
35         ...
36     };
37 }
38 ...

```

2:26 Nested Pure Operation-Based CRDTs

■ **Listing 8** User API for local mutations to DFS CRDT, allowing simple modification of the DFS meta-data.

```
1  CreateUser(with_admin_rights: boolean) {
2    const id = this.getUID();
3    this.performOp("CreateUser", [with_admin_rights, id]);
4    return id;
5  };
6
7  CreateGroup() {
8    const id = this.getUID();
9    this.performNestedOp("update", [{ key: "groups", op: "update" },
10   { key: id, op: "update" }],
11   { key: "created", op: "write" }], [FLAG_TRUE]);
12   return id;
13 };
14
15 CreateFile(userId: UserID, groupId: GroupID) {
16   const id = this.getUID();
17   this.performOp("CreateFile", [userId, groupId, id]);
18   return id;
19 }
20 ...
```

■ **Listing 9** Example test code for the DFS CRDT, which creates a new admin user, a new group, adds the user to a group, and then creates and writes a file with this new user.

```
1  test() {
2    const userId = this.CreateUser(true);
3    const groupId = this.CreateGroup();
4
5    this.performOp("AssignUserToGroup", [userId, groupId, userId]);
6
7    const fileId = this.CreateFile(userId, groupId);
8    this.performOp("WriteFile", [userId, fileId]);
9
10 }
```

Multi-Graded Featherweight Java

Riccardo Bianchini  

DIBRIS, University of Genova, Italy

Francesco Dagnino  

DIBRIS, University of Genova, Italy

Paola Giannini  

DiSSTE, University of Eastern Piedmont, Vercelli, Italy

Elena Zucca  

DIBRIS, University of Genova, Italy

Abstract

Resource-aware type systems statically approximate not only the expected result type of a program, but also the way external resources are used, e.g., how many times the value of a variable is needed. We extend the type system of Featherweight Java to be resource-aware, parametrically on an arbitrary *grade algebra* modeling a specific usage of resources. We prove that this type system is *sound* with respect to a resource-aware version of reduction, that is, a well-typed program has a reduction sequence which does not get stuck due to resource consumption. Moreover, we show that the available grades can be *heterogeneous*, that is, obtained by combining grades of different kinds, via a minimal collection of homomorphisms from one kind to another. Finally, we show how grade algebras and homomorphisms can be specified as Java classes, so that grade annotations in types can be written in the language itself.

2012 ACM Subject Classification Theory of computation → Type structures

Keywords and phrases Graded modal types, Java

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.3

Funding This work was partially funded by the MUR project “T-LADIES” (PRIN 2020TL3X8X) and has the financial support of the University of Eastern Piedmont.

Acknowledgements We thank the anonymous referees for their useful suggestions.

1 Introduction

Recently, a considerable amount of research [25, 7, 2, 14, 15, 23, 8, 11] has been devoted to type systems allowing reasoning about resource usage. In *(type-and-)coeffect systems*, the typing judgment takes the shape $x_1 :_{r_1} T_1, \dots, x_n :_{r_n} T_n \vdash e : T$, where the *coeffect (grade)* r_i models how variable x_i is used in e . For instance, coeffects of shape $r ::= 0 \mid 1 \mid \omega$ trace when a variable is either not used, or used at most once, or used in an unrestricted way, respectively. In this way, functions, e.g., $\lambda x:\text{int}.5$, $\lambda x:\text{int}.x$, and $\lambda x:\text{int}.x + x$, which have the same type in the simply-typed lambda calculus, can be distinguished by adding coeffect annotations: $\lambda x:\text{int}[0].5$, $\lambda x:\text{int}[1].x$, and $\lambda x:\text{int}[\omega].x + x$. Other examples are exact usage (coeffects are natural numbers), and privacy levels. *Graded modal types* go further, by decorating types themselves with grades, in order to specify how *the result of an expression* should be used. In the different proposals in literature, grades have a similar algebraic structure, basically a semiring specifying *sum* $+$, *multiplication* \cdot , and 0 and 1 constants, and some kind of order relation. Here, we will assume a variant of this notion called *grade algebra*.

Resource-aware typing has been exploited in a fully-fledged programming language in Granule [23], a functional language equipped with graded modal types, hence allowing the programmer to write function declarations similar to those above. In Granule, different



© Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 3; pp. 3:1–3:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

kinds of coeffacts can be used at the same time, including naturals for exact usage, privacy levels, intervals, infinity, and products of coeffacts; however, available grades are fixed in the language. The initial objective of the work presented here was to study a similar support for Java-like languages, by introducing, in a variant of Featherweight Java (FJ) [19], types decorated with grades. Moreover, we wanted these grades to be taken, parametrically, in an arbitrary grade algebra; even more, we did not want this grade algebra to be fixed as in Granule, but to be extendable by the programmer with user-defined grades, by relying on the inheritance mechanism of OO languages. In the quest for such goals, we came up with several ideas which are novel, to our knowledge, with respect to the literature on resource-aware type systems, as detailed in the outline of contributions given below.

Resource-aware parametric FJ reduction. Given a resource-aware type system, we would like to prove that typing overapproximates the use of resources. However, resource usage is not modeled in standard operational semantics; for this reason, [8] proposed an instrumented operational semantics¹ and proved a soundness theorem showing correct accounting of resource usage. Inspired by this work, we define a *resource-aware semantics* for FJ, parametric on an arbitrary grade algebra, which tracks how much each available resource is consumed at each step, and is stuck when the needed amount of a resource is not available. Differently from [8], the semantics is given *independently* from the type system, as is the standard approach in calculi. That is, the aim is also to provide a simple purely semantic model which takes into account usage of resources. The resource-aware reduction is sound with respect to the standard reduction, but clearly not complete, since a reduction step allowed in the standard semantics could be impossible due to resource consumption.

Graded FJ. After defining the resource-aware calculus, we define the resource-aware type system. That is, types are decorated with grades, allowing the programmer to specify how a variable, a field or the result of a method should be used, e.g., how many times. Our approach is novel with respect to that generally used in the literature on graded modal types. Notably, in such works the production of types is $T ::= \dots \mid T^r$, that is, grade decorations can be arbitrarily nested. Correspondingly, the syntax includes an explicit *box* construct, which transforms a term of type T into a term of type T^r , through a *promotion* rule which multiplies the context with r , and a corresponding unboxing mechanism. Here, we prefer a much lighter approach, likely more convenient for Java-like languages, where the syntax of terms is not affected. The production for types is $T ::= C^r$, that is, all types (here only class names) are (once) graded; in contexts, types are non-graded, and grades are used as coeffacts, leading to a judgment of shape $x_1 :_{r_1} C_1, \dots, x_n :_{r_n} C_n \vdash e : C^r$. Finally, since there is no boxing/unboxing, there is no explicit promotion rule, but different grades can be assigned to an expression, assuming different coeffact contexts. We prove a soundness theorem, stating that the graded type system overapproximates resource usage, hence guaranteeing soundness, and, as a consequence, completeness with respect to standard reduction for well-typed programs.

Combining grades. The next matter is how to make the language *multi-graded*, in the sense that the programmer can use grades of different kinds, e.g., both natural numbers and privacy levels. This poses the problem of defining the result when grades of different kinds should

¹ Subsequently the model of [8] was used, in [21], to trace reference counting for uniqueness.

be combined by the type system. This issue has been considered in the Granule language [23], where, however, the available kinds of grades are fixed, hence can be combined in an ad-hoc way. We would like to have much more flexibility, that is, to allow the programmer to define grades to be added to those already available, very much in the same way a Java programmer can define her/his own class of exceptions. To this end, we define a construction which, given a family of grade algebras and a family of homomorphisms, leads to a unique grade algebra of *heterogeneous grades*. This allows a modular approach, in the sense that the developed meta-theory, including the proof of results, applies to this case as well.

Grades as Java expressions. Finally, we consider the issue of providing linguistic support to specify the desired grade algebras and homomorphisms. Of course this could be done by using an ad-hoc configuration language. However, we believe an interesting solution is that the grade annotations could be written themselves in Java, again analogously to what happens with exceptions. We describe how Java classes corresponding to grade algebras and homomorphisms could be written, providing some examples.

A preliminary step towards the results described in the current paper is [3], which proposes a first version of the type system with only coefficients (types are not graded), and a rudimentary version of the construction described above where combining coefficients of different kinds leads to the trivial coefficient.

In Section 2 we formally define grade algebras and related notions. In Section 3 we define the parametric resource-aware reduction for FJ, and in Section 4 the parametric resource-aware type system, proving its soundness. Section 5 defines the construction of the grade algebra of heterogeneous grades, and Section 6 illustrates how to express grade algebras and homomorphisms in Java. Finally, Section 7 surveys related work and Section 8 summarizes the contributions, and outlines future work. Omitted proofs can be found in [4].

2 Algebraic preliminaries

In this section we introduce the algebraic structures we will use throughout the paper. The core of our work is *grades*, namely, annotations in the code expressing how or how much resources are used by the program. As we will see, we need some operations to properly combine grades in the resource-aware semantics and in the typing rules, hence we will assume grades to form an algebraic structure called *grade algebra* defined below.

► **Definition 1** (Grade algebra). *A grade algebra is a tuple $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ such that:*

- $\langle |R|, \preceq \rangle$ is a partially ordered set;
 - $\langle |R|, +, \mathbf{0} \rangle$ is a commutative monoid;
 - $\langle |R|, \cdot, \mathbf{1} \rangle$ is a monoid;
- and the following axioms are satisfied:
- $r \cdot (s + t) = r \cdot s + r \cdot t$ and $(s + t) \cdot r = s \cdot r + t \cdot r$, for all $r, s, t \in |R|$;
 - $r \cdot \mathbf{0} = \mathbf{0}$ and $\mathbf{0} \cdot r = \mathbf{0}$, for all $r \in |R|$;
 - if $r \preceq r'$ and $s \preceq s'$ then $r + s \preceq r' + s'$ and $r \cdot s \preceq r' \cdot s'$, for all $r, r', s, s' \in |R|$;
 - $\mathbf{0} \preceq r$, for all $r \in |R|$.

Essentially, a grade algebra is an ordered semiring, that is, a semiring together with a partial order relation on its carrier which makes addition and multiplication monotonic with respect to it. We further require the zero of the semiring to be the least element of the partial order. Our definition is a slight variant of others proposed in literature [7, 15, 22, 2, 14, 1, 23, 8, 27]. In particular, the partial order models overapproximation in

the usage of resources, and allows flexibility, for instance we can have different usage in the branches of an if-then-else construct. The fact that the zero is the least element means that, in particular, overapproximation can add unused variables, making the calculus *affine*.

► **Example 2.**

1. The semiring $\mathbf{Nat} = \langle \mathbb{N}, \leq, +, \cdot, 0, 1 \rangle$ of natural numbers with the natural order and usual arithmetic operations is a grade algebra.
2. The *affinity* grade algebra $\langle \{0, 1, \infty\}, \leq, +, \cdot, 0, 1 \rangle$ is obtained from the previous one by identifying all natural numbers greater than 1.
3. The trivial semiring \mathbf{Triv} , whose carrier is a singleton set $|\mathbf{Triv}| = \{\infty\}$, the partial order is the equality, addition and multiplication are defined in the trivial way and $\mathbf{0}_{\mathbf{Triv}} = \mathbf{1}_{\mathbf{Triv}} = \infty$, is a grade algebra.
4. The semiring $\mathbf{R}_{\geq 0}^{\infty} = \langle [0, \infty], \leq, +, \cdot, 0, 1 \rangle$ of extended non-negative real numbers with usual order and operations, extended to ∞ in the expected way, is a grade algebra.
5. A distributive lattice $\mathbf{L} = \langle |\mathbf{L}|, \leq, \vee, \wedge, \perp, \top \rangle$, where \vee and \wedge denote join and meet operations and \perp and \top the bottom and the top element, respectively, is a grade algebra.
6. Given grade algebras $R = \langle |R|, \preceq_R, +_R, \cdot_R, \mathbf{0}_R, \mathbf{1}_R \rangle$ and $S = \langle |S|, \preceq_S, +_S, \cdot_S, \mathbf{0}_S, \mathbf{1}_S \rangle$, the *product* $R \times S = \langle \{ \langle r, s \rangle \mid r \in |R| \wedge s \in |S| \}, \preceq, +, \cdot, \langle \mathbf{0}_R, \mathbf{0}_S \rangle, \langle \mathbf{1}_R, \mathbf{1}_S \rangle \rangle$, where operations are the pairwise application of the operations for R and S , is a grade algebra.
7. Given a grade algebra $R = \langle |R|, \preceq_R, +_R, \cdot_R, \mathbf{0}_R, \mathbf{1}_R \rangle$, as in [23] we define $\mathbf{Ext} R = \langle |R| + \{\infty\}, \preceq, +, \cdot, \mathbf{0}_R, \mathbf{1}_R \rangle$ where \preceq extends \preceq_R by adding $r \preceq \infty$ for all $r \in |\mathbf{Ext} R|$ and $+$ and \cdot extend $+_R$ and \cdot_R by $r + \infty = \infty + r = \infty$, for all $r \in |\mathbf{Ext} R|$, and $r \cdot \infty = \infty \cdot r = \infty$, for all $r \in |\mathbf{Ext} R|$ with $r \neq \mathbf{0}_R$, and $\mathbf{0}_R \cdot \infty = \infty \cdot \mathbf{0}_R = \mathbf{0}_R$. Then, $\mathbf{Ext} R$ is a grade algebra.

A homomorphism of grade algebras $f: R \rightarrow S$ is a monotone function $f: \langle |R|, \preceq_R \rangle \rightarrow \langle |S|, \preceq_S \rangle$ between the underlying partial orders, which preserves the semiring structure, that is, satisfies the following equations:

- $f(\mathbf{0}_R) = \mathbf{0}_S$ and $f(r +_R s) = f(r) +_S f(s)$, for all $r, s \in |R|$;
- $f(\mathbf{1}_R) = \mathbf{1}_S$ and $f(r \cdot_R s) = f(r) \cdot_S f(s)$, for all $r, s \in |R|$.

Grade algebras and their homomorphisms form a category denoted by \mathbf{GrAlg} .

Consider a grade algebra R . Then, we can define functions $\zeta_R: |R| \rightarrow |\mathbf{Triv}|$ and $\iota_R: |\mathbf{Nat}| \rightarrow |R|$ as follows:

$$\zeta_R(r) = \infty \quad \iota_R(m) = \begin{cases} \mathbf{0}_R & \text{if } m = 0 \\ \iota_R(n) +_R \mathbf{1}_R & \text{if } m = n + 1 \end{cases}$$

Roughly, ζ_R maps every element of R to ∞ , while ι_R maps a natural number n to the sum in R of n copies of $\mathbf{1}_R$. We can easily check that both these functions give rise to homomorphisms $\zeta_R: R \rightarrow \mathbf{Triv}$ and $\iota_R: \mathbf{Nat} \rightarrow R$. This is straightforward for ζ_R , while for ι_R follows by arithmetic induction. Then, we can prove the following result.

► **Proposition 3.** *The following facts hold:*

1. \mathbf{Nat} is the initial object in \mathbf{GrAlg} ;
2. \mathbf{Triv} is the terminal object in \mathbf{GrAlg} .

Another kind of objects we will work with are maps assigning grades to variables. These inherit a nice algebraic structure from the one of the underlying grade algebra.

Assume a grade algebra $R = \langle |R|, \preceq, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ and a set X . The set of functions from X to $|R|$ carries a partially ordered commutative monoid structure given by the pointwise extension of the additive structure of R . That is, given $\gamma, \gamma': X \rightarrow |R|$, we define $\gamma \preceq \gamma'$

iff, for all $x \in X$, $\gamma(x) \preceq \gamma'(x)$, and $(\gamma + \gamma')(x) = \gamma(x) + \gamma'(x)$ and $\hat{\mathbf{0}}(x) = \mathbf{0}$, for all $x \in X$. Moreover, we can define a *scalar multiplication*, combining elements of $|R|$ and a function $\gamma: X \rightarrow |R|$; indeed, we set $(r \cdot \gamma)(x) = r \cdot \gamma(x)$, for all $r \in |R|$ and $x \in X$. It is easy to see that this operation turns the partially ordered commutative monoid of functions from X to $|R|$ into a partially ordered R -module.

The *support* of a function $\gamma: X \rightarrow |R|$ is the set $\mathbf{S}(\gamma) = \{x \in X \mid \gamma(x) \neq \mathbf{0}\}$. Denote by R^X the set of functions $\gamma: X \rightarrow |R|$ with finite support. The partial order and operations defined above can be safely restricted to R^X , noting that $\mathbf{S}(\hat{\mathbf{0}}) = \emptyset$, $\mathbf{S}(\gamma + \gamma') \subseteq \mathbf{S}(\gamma) \cup \mathbf{S}(\gamma')$ and $\mathbf{S}(r \cdot \gamma) \subseteq \mathbf{S}(\gamma)$. Therefore, R^X carries a partially ordered R -module structure as well.

As we will see in Section 4, *coeffect contexts* are (representations of) functions in R^X , with X set of variables. The fact that coeffect contexts form a module has been firstly noted in [22, 27], and fully formalized in [5], which also shows a *non-structural* example. That is, a module different from R^X described above, used in the present paper and mostly in the literature, is needed, where operations on coeffect contexts are not pointwise.

3 Resource-aware semantics

Standard operational models do not say anything about resources used by the computation. To address this problem, we follow an approach similar to that in [8], that is, we define an *instrumented* semantics which keeps track of resource usage, hence, in particular, it gets stuck if some needed resource is insufficient. However, unlike [8], the definition of our resource-aware semantics, though parameterized on a grade algebra, is given *independently* of the graded type system, as is the standard approach in calculi; in the next section, we will show how the graded type system actually overapproximates resource usage, hence guarantees soundness. As will be detailed in the following, the resource-aware semantics is non-deterministic, in the sense that, when a resource is needed, it can be consumed in different ways; hence, soundness is *soundness-may*, meaning that there is a reduction which does not get stuck because of standard typing errors or resource consumption.

Reference calculus. The calculus is a variant of FJ [19]. The syntax is reported in the top section of Figure 1. We write es as a metavariable for e_1, \dots, e_n , $n \geq 0$, and analogously for other sequences. We assume *variables* x, y, z, \dots , *class names* C, D , *field names* f , and *method names* m . Types are distinct from class names to mean that they could be extended to include other types, e.g., primitive types. In addition to the standard FJ constructs, we have a block expression, consisting of a local variable declaration, and a body.

The semantics is defined differently from the original one; that is, reduction is defined on *configurations* $e|\rho$, where ρ is an *environment*, a finite map from variables into values. In this way, variable occurrences are replaced one at a time by their value in the environment, rather than once and for all. This definition can be easily shown to be equivalent to the original one, and is convenient for our aims since, in this presentation, free variables in an expression can be naturally seen as *resources* which are consumed each time a variable occurrence is *used* (replaced by its value) during execution. In other words, this semantics can be naturally *instrumented* by adding grades expressing the “cost” of resource consumption, as we will do in Figure 2. Apart from that, the rules are straightforward; only note that, in rules (INVK) and (BLOCK), parameters (including **this**) and local variable are renamed to fresh variables, to avoid clashes. Single contextual rules are given, rather than defining evaluation contexts, to be uniform with the instrumented version, where this presentation is more convenient.

3:6 Multi-Graded Featherweight Java

e	$::=$	$x \mid e.f \mid \mathbf{new} C(es) \mid e.m(es) \mid \{T x = e; e'\}$	expression
T	$::=$	C	type (class name)
v	$::=$	$\mathbf{new} C(vs)$	value

$$\text{(VAR)} \quad \frac{}{x|\rho \rightarrow v|\rho} \quad \rho(x) = v$$

$$\text{(FIELD-ACCESS)} \quad \frac{\text{fields}(C) = T_1 f_1; \dots T_n f_n;}{\mathbf{new} C(v_1, \dots, v_n).f_i|\rho \rightarrow v_i|\rho} \quad i \in 1..n$$

$$\text{(INVK)} \quad \frac{}{v_0.m(v_1, \dots, v_n)|\rho \rightarrow e[y_0/\mathbf{this}][y_1/x_1 \dots y_n/x_n]|\rho'} \quad \begin{array}{l} v_0 = \mathbf{new} C(_) \\ \text{mbody}(C, m) = \langle x_1 \dots x_n, e \rangle \\ y_0, \dots, y_n \notin \text{dom}(\rho) \\ \rho' = \rho, y_0 \mapsto v_0, \dots, y_n \mapsto v_n \end{array}$$

$$\text{(BLOCK)} \quad \frac{}{\{C x = v; e\}|\rho \rightarrow e[y/x]|\rho, y \mapsto v} \quad y \notin \text{dom}(\rho)$$

$$\text{(FIELD-ACCESS-CTX)} \quad \frac{e|\rho \rightarrow e'|\rho'}{e.f|\rho \rightarrow e'.f|\rho'}$$

$$\text{(NEW-CTX)} \quad \frac{e_i|\rho \rightarrow e'_i|\rho'}{\mathbf{new} C(v_1, \dots, v_{i-1}, e_i, \dots, e_n)|\rho \rightarrow \mathbf{new} C(v_1, \dots, v_{i-1}, e'_i, \dots, e_n)|\rho'}$$

$$\text{(INVK-RCV-CTX)} \quad \frac{e_0|\rho \rightarrow e'_0|\rho'}{e_0.m(e_1, \dots, e_n)|\rho \rightarrow e'_0.m(e_1, \dots, e_n)|\rho'}$$

$$\text{(INVK-ARG-CTX)} \quad \frac{e_i|\rho \rightarrow e'_i|\rho'}{v_0.m(v_1, \dots, v_{i-1}, e_i, \dots, e_n)|\rho \rightarrow v_0.m(v_1, \dots, v_{i-1}, e'_i, \dots, e_n)|\rho'}$$

$$\text{(BLOCK-CTX)} \quad \frac{e_1|\rho \rightarrow e'_1|\rho'}{\{C x = e_1; e_2\}|\rho \rightarrow \{C x = e'_1; e_2\}|\rho'}$$

■ **Figure 1** Syntax and standard reduction.

To be concise, the class table is abstractly modeled as follows, omitting its (standard) syntax:

- $\text{fields}(C)$ gives, for each class C , the sequence $T_1 f_1; \dots T_n f_n$; of its fields, assumed to have distinct names, with their types;
- $\text{mbody}(C, m)$ gives, for each method m of class C , its parameters and body.

Instrumented reduction. This reduction uses *grades*, ranged over by r, s, t , assumed to form a grade algebra, specifying a *partial order* \preceq , a *sum* $+$, a *multiplication* \cdot , and constants $\mathbf{0}$ and $\mathbf{1}$, satisfying some axioms, as detailed in Definition 1 of Section 2.

In order to keep track of usage of resources, parametrically on a given grade algebra, we *instrument* reduction as follows.

- The environment associates, to each resource (variable), besides its value, a grade modeling its *allowed usage*.

- Moreover, the reduction relation is *graded*, that is, indexed by a grade r , meaning that it aims at producing a value to be used (at most) r times, or, in more general (non-quantitative) terms, to be used (at most) with grade r .
- The grade of a variable in the environment decreases, each time the variable is used, of the amount specified in the reduction grade².
- Of course, this can only happen if the current grade of the variable *can* be reduced of such an amount; otherwise the reduction is stuck.

Before giving the formal definition, we show some simple examples of reductions, considering the grade algebra of naturals of Example 2(1), tracking how many times a resource is used.

► **Example 4.** Assume the following classes:

```
class A {}
class Pair {A first; A second}
```

We write v_{pair} as an abbreviation for $\text{new Pair}(\text{new A}(), \text{new A}())$.

```
{A a = [new A()]4; {Pair p = [new Pair(a, a)]2; new Pair(p.first, p.second)}}|∅ →1
{Pair p = [new Pair(a, a)]2; new Pair(p.first, p.second)}|a ↦ ⟨new A(), 4⟩ →1
{Pair p = [new Pair(new A(), a)]2; new Pair(p.first, p.second)}|a ↦ ⟨new A(), 2⟩ →1
{Pair p = [new Pair(new A(), new A())]2; new Pair(p.first, p.second)}|a ↦ ⟨new A(), 0⟩ →1
new Pair(p.first, p.second)|a ↦ ⟨new A(), 0⟩, p ↦ ⟨vpair, 2⟩ →1
new Pair(vpair.first, p.second)|a ↦ ⟨new A(), 0⟩, p ↦ ⟨vpair, 1⟩ →1
new Pair(new A(), p.second)|a ↦ ⟨new A(), 0⟩, p ↦ ⟨vpair, 1⟩ →1
new Pair(new A(), vpair.second)|a ↦ ⟨new A(), 0⟩, p ↦ ⟨vpair, 0⟩ →1
vpair|a ↦ ⟨new A(), 0⟩, p ↦ ⟨vpair, 0⟩
```

In the example, the top-level reduction is graded 1, meaning that a single value is produced. Subterms are annotated with the grade of their reduction. For instance, the outer initialization expression is annotated 4, meaning that its result can be used (at most) 4 times. To lighten the notation, in this example we omit the index 1. A local variable introduced in a block is added³ as another available resource in the environment, with the value and the grade of its initialization expression; for instance, the outer local variable is added with grade 4. When evaluating the inner initialization expression, which is reduced with grade 2, each time the variable a is used its grade in the environment is decremented by 2.

It is important to notice that the annotations in subterms are *not* type annotations. Except those in arguments of constructor invocation, explained below, annotations are only needed to ensure that reduction of a subterm happens at each step with the same grade, see the formal definition below. We plan to investigate in future work a big-step formulation which would not need such an artifice. In the example above, we have chosen for the reduction of subterms the minimum grade allowing to perform the top-level reduction. We could have chosen any greater grade; instead, with a strictly lower grade, the reduction would be stuck.

As anticipated, in a constructor invocation $\text{new } C([e_1]_{r_1}, \dots, [e_n]_{r_n})$, the annotation r_i plays a special role: intuitively, it specifies that the object to be constructed should contain r_i copies of that field. Formally, this is reflected by the reduction grade of the subterm e_i , which must be exactly $r \cdot r_i$, if r is the reduction grade of the object, specifying how many copies of it the reduction is constructing. Correspondingly, an access to the field can be used (at most) $r \cdot r_i$ times. This is illustrated by the following variant of the previous example.

² More precisely, the reduction grade acts as a lower bound for this amount, see comment to rule (VAR).

³ Modulo renaming to avoid clashes, omitted in the example for simplicity.

► **Example 5.** Consider the term

$$\{A \ a = [\text{new } A()]_4; \{\text{Pair } p = [\text{new } \text{Pair}(a, a)]_2; \text{new } \text{Pair}([\text{p.first}]_2, p.\text{second})\}\}$$

As highlighted in grey, the first argument of the constructor invocation which is the body of the inner block is now annotated with 2, meaning that the resulting object should have “two copies” of the field. As a consequence, the expression `p.first` should be reduced with grade 2, as shown below, where $v_{\text{pair}} = \text{new } \text{Pair}(\text{new } A(), \text{new } A())$, the first four reduction steps are as in Example 4 and we explicitly write some annotations 1 for clarity

$$\begin{aligned} & \{A \ a = [\text{new } A()]_4; \{\text{Pair } p = [\text{new } \text{Pair}([a]_1, a)]_2; \text{new } \text{Pair}([\text{p}]_1.\text{first}]_2, p.\text{second})\}\} | \emptyset \rightarrow_1^* \\ & \text{new } \text{Pair}([\text{p}]_1.\text{first}]_2, p.\text{second}) | a \mapsto \langle \text{new } A(), 0 \rangle, p \mapsto \langle v_{\text{pair}}, 2 \rangle \rightarrow_1 \\ & \text{new } \text{Pair}([v_{\text{pair}}]_1.\text{first}]_2, p.\text{second}) | a \mapsto \langle \text{new } A(), 0 \rangle, p \mapsto \langle v_{\text{pair}}, 1 \rangle \quad \text{STUCK} \end{aligned}$$

Reduction of the subterm in grey, aiming at constructing a value (`new A()`) which can be used twice, is stuck, since we cannot obtain two copies of `new A()` from the field `first` of the object v_{pair} . If we choose, instead, to reduce the occurrence of `p` to be used twice, then we get the following reduction, where again we omit steps which are as before:

$$\begin{aligned} & \{A \ a = [\text{new } A()]_4; \{\text{Pair } p = [\text{new } \text{Pair}([a]_1, a)]_2; \text{new } \text{Pair}([\text{p}]_2.\text{first}]_2, p.\text{second})\}\} | \emptyset \rightarrow_1^* \\ & \text{new } \text{Pair}([\text{p}]_2.\text{first}]_2, p.\text{second}) | a \mapsto \langle \text{new } A(), 0 \rangle, p \mapsto \langle v_{\text{pair}}, 2 \rangle \rightarrow_1 \\ & \text{new } \text{Pair}([v_{\text{pair}}]_2.\text{first}]_2, p.\text{second}) | a \mapsto \langle \text{new } A(), 0 \rangle, p \mapsto \langle v_{\text{pair}}, 0 \rangle \rightarrow_1 \\ & \text{new } \text{Pair}(\text{new } A()]_2, p.\text{second}) | a \mapsto \langle \text{new } A(), 0 \rangle, p \mapsto \langle v_{\text{pair}}, 0 \rangle \quad \text{STUCK} \end{aligned}$$

In this case, the reduction is stuck since we consumed all the available copies of `p` to produce two copies of the field `first`, so now we cannot reduce `p.second`. To obtain a non-stuck reduction, we should choose to reduce the initialization expression of `p` with index 3, hence that of `a` with index 6. To complete the construction of the `Pair`, that is, to get a non-stuck reduction, we should have 3 copies of `p` and therefore 6 copies of `a`.

The formal definition of the instrumented semantics is given in Figure 2. To make the notation lighter, we use the same metavariables of the standard semantics in Figure 1. As explained above, reduction is defined on annotated terms. Notably, in each construct, the subterms which are reduced in contextual rules are annotated, so that their reduction always happens with a fixed grade.

In rule (VAR), which is the key rule where resources are consumed, a variable occurrence is replaced by the associated value in the environment, and its grade s decreases to s' , burning a non-zero amount r' of resources which has to be at least the reduction grade. The side condition $r' + s' \preceq s$ ensures that the initial grade of the variable suffices to cover both the consumed grade and the residual grade. To show why the amount of resource consumption should be non-zero, consider, e.g., the following variant of Example 4:

$$\{A \ a = [\text{new } A()]_4; \{\text{Pair } p = [\text{new } \text{Pair}(a, a)]_0; \text{new } \text{Pair}(a, a)\}\} | \emptyset$$

The local variable `p` is never used in the body of the block, so it makes sense for its initialization expression to be reduced with grade 0, since execution needs no copies of the result. Yet, the expression *needs to be reduced*, and to produce its useless result two copies of `a` are consumed; in a sense, they are wasted. However, such resource usage is tracked, whereas it would be lost if decrementing by 0. Removing the non-zero requirement would lead to a variant of resource-aware reduction where usage of resource which are useless to construct the final result is not tracked.

In rule (FIELD-ACCESS), the reduction grade should be (overapproximated by) the multiplication of the grade of the receiver with that of the field (constructor argument). Indeed, the former specifies how many copies of the object we have and the latter how many copies

$$\begin{aligned}
e &::= x \mid [e]_r \cdot f \mid \mathbf{new} C([e_1]_{r_1}, \dots, [e_n]_{r_n}) \mid && \text{(annotated) expression} \\
& \quad [e_0]_{r_0} \cdot m([e_1]_{r_1}, \dots, [e_n]_{r_n}) \text{es} \mid \{T \ x = [e]_r; e'\} \\
v &::= \mathbf{new} C([v_1]_{r_1}, \dots, [v_n]_{r_n}) && \text{(annotated) value}
\end{aligned}$$

$$\begin{aligned}
(\text{VAR}) \quad & \frac{}{x|\rho, x \mapsto \langle v, s \rangle \rightarrow_r v|\rho, x \mapsto \langle v, s' \rangle} \quad r \leq r' \neq \mathbf{0} \quad s' + r' \leq s \\
(\text{FIELD-ACCESS}) \quad & \frac{\mathbf{new} C([v_1]_{r_1}, \dots, [v_n]_{r_n}) \cdot f_i|\rho \rightarrow_s v_i|\rho}{\mathbf{new} C([v_1]_{r_1}, \dots, [v_n]_{r_n}) \cdot f_i|\rho \rightarrow_s v_i|\rho} \quad \begin{array}{l} \text{fields}(C) = T_1 f_1; \dots T_n f_n; \\ i \in 1..n \\ s \leq r \cdot r_i \end{array} \\
(\text{INVK}) \quad & \frac{[v_0]_{r_0} \cdot m([v_1]_{r_1}, \dots, [v_n]_{r_n})|\rho \rightarrow_r e[y_0/\mathbf{this}][y_1/x_1 \dots y_n/x_n]|\rho'}{[v_0]_{r_0} \cdot m([v_1]_{r_1}, \dots, [v_n]_{r_n})|\rho \rightarrow_r e[y_0/\mathbf{this}][y_1/x_1 \dots y_n/x_n]|\rho'} \quad \begin{array}{l} v_0 = \mathbf{new} C(_) \\ \mathbf{mbody}(C, m) = \langle x_1 \dots x_n, e \rangle \\ y_0, \dots, y_n \notin \mathbf{dom}(\rho) \\ \rho' = \rho, y_0 \mapsto \langle v_0, r_0 \rangle, \dots, y_n \mapsto \langle v_n, r_n \rangle \end{array} \\
(\text{BLOCK}) \quad & \frac{}{\{C \ x = [v]_r; e\}|\rho \rightarrow_s e[y/x]|\rho, y \mapsto \langle v, r \rangle} \quad y \notin \mathbf{dom}(\rho) \\
(\text{FIELD-ACCESS-CTX}) \quad & \frac{e|\rho \rightarrow_r e'|\rho'}{[e]_r \cdot f|\rho \rightarrow_s [e']_r \cdot f|\rho'} \\
(\text{NEW-CTX}) \quad & \frac{e_i|\rho \rightarrow_{r \cdot r_i} e'_i|\rho'}{\mathbf{new} C([v_1]_{r_1}, \dots, [v_{i-1}]_{r_{i-1}}, [e_i]_{r_i}, \dots, [e_n]_{r_n})|\rho \rightarrow_r \mathbf{new} C([v_1]_{r_1}, \dots, [v_{i-1}]_{r_{i-1}}, [e'_i]_{r_i}, \dots, [e_n]_{r_n})|\rho'} \\
(\text{INVK-RCV-CTX}) \quad & \frac{e_0|\rho \rightarrow_{r_0} e'_0|\rho'}{[e_0]_{r_0} \cdot m([e_1]_{r_1}, \dots, [e_n]_{r_n})|\rho \rightarrow_r [e'_0]_{r_0} \cdot m([e_1]_{r_1}, \dots, [e_n]_{r_n})|\rho'} \\
(\text{INVK-ARG-CTX}) \quad & \frac{e_i|\rho \rightarrow_{r_i} e'_i|\rho'}{[e_0]_{r_0} \cdot m([v_1]_{r_1}, \dots, [v_{i-1}]_{r_{i-1}}, [e_i]_{r_i}, \dots, [e_n]_{r_n})|\rho \rightarrow_r [e_0]_{r_0} \cdot m([v_1]_{r_1}, \dots, [v_{i-1}]_{r_{i-1}}, [e'_i]_{r_i}, \dots, [e_n]_{r_n})|\rho'} \\
(\text{BLOCK-CTX}) \quad & \frac{e_1|\rho \rightarrow_s e'_1|\rho'}{\{C \ x = [e_1]_s; e_2\}|\rho \rightarrow_r \{C \ x = [e'_1]_s; e_2\}|\rho'}
\end{aligned}$$

■ **Figure 2** Instrumented reduction.

of the field each of such objects has; thus, their product provides an upper bound to the grade of the resulting value. Note that, in this way, some reductions could be *forbidden*. For instance, taking the grade algebra of naturals, an access to a field whose value can be used 3 times, of an object reduced with grade 2, can be reduced with grade (at most) 6. Another more significant example is given in the following, taking the grade algebra of *privacy levels*.

Rule (INVK) adds each method parameter, including **this**, as available resource in the environment, modulo renaming with a fresh variable to avoid clashes. The associated value and grade are that of the corresponding argument. Rule (BLOCK) is exactly analogous, apart that only one variable is added.

Coming to contextual rules, the reduction grade of the subterm is that of the corresponding annotation, so that all steps happen with a fixed grade. The only exception is rule (NEW-CTX), where, symmetrically to rule (FIELD-ACCESS), the reduction grade for subterms should be the multiplication of the reduction grade of the object with the annotation of the field (constructor argument), capturing the intuition that the latter specifies the grade of the field for a single copy of the object. For instance, taking the grade algebra of naturals, to obtain an object which can be used twice, with a field which can be used 3 times, the value of such field should be an object which can be used 6 times.

Note that, besides the standard typing errors such as looking for a missing method or field, reduction graded r can get stuck since either rule (VAR) cannot be applied since the side conditions do not hold, or rule (FIELD-ACCESS) cannot be applied since the side condition $s \preceq r \cdot r_i$ does not hold. Informally, either some resource (variable) is exhausted, that is, can no longer be replaced by its value, or some field of some object cannot be extracted. It is also important to note that the instrumented reduction is non-deterministic, due to rule (VAR).

In the grade algebra used in the previous example, grades model *how many times* resources are used. However, grades can also model a non-quantitative⁴ knowledge, that is, track possible *modes* in which a resource can be used, or, in other words, possible *constraints* on how it could be used. A typical example of this situation are *privacy levels*, which can be formalized similarly to what is done in [1], as described below.

► **Example 6.** Starting from any distributive semilattice lattice L , like in Example 2(5), define $L_0 = \langle |L_0|, \leq_0, \vee_0, \wedge_0, 0, \top \rangle$, where $|L_0| = |L| + \{0\}$ with $0 \leq_0 x$, $x \vee_0 0 = 0 \vee_0 x = x$ and $x \wedge_0 0 = 0 \wedge_0 x = 0$, for all $x \in |L|$; on elements of $|L|$ the order and the operations are those of L . That is, we assume that the privacy levels form a distributive semilattice with order representing “decreasing privacy”, and we add a grade 0 modeling “non-used”. The simplest instance consists of just two privacy levels, that is, $0 \preceq \text{private} \preceq \text{public}$. Sum is the join, meaning that we obtain a privacy level which is less restrictive than both: for instance, a variable which is used as `public` in a subterm, and as `private` in another, is overall used as `public`. Multiplication is the meet, meaning that we obtain a privacy level which is more restrictive than both: for instance, an access to a field whose value has been obtained in `public` mode, of an object reduced in `private` mode, is reduced in `private` mode⁵. Note that exactly the same structure could be used to model, e.g., rather than privacy levels, modifiers `readonly` and `mutable` in an imperative setting, corresponding to forbid field assignment and no restrictions, respectively. The following examples illustrates the use of such grade algebra. We write `priv` and `pub` for short, and classes `A` and `Pair` are as in the previous examples.

1. Let $e_1 = \{A\ y = [\text{new } A()]_{\text{pub}}; \{A\ x = [y]_{\text{priv}}; x\}\}$ and p_- be either `pub` or `priv`, e_1 starting with the empty environment reduces with grade `private` as follows:

$$\begin{aligned}
e_1|\emptyset &\xrightarrow{\text{priv}} \{A\ x = [y]_{\text{priv}}; x\}|y \mapsto \langle \text{new } A(), \text{pub} \rangle \text{ with (BLOCK)} \\
&\xrightarrow{\text{priv}} \{A\ x = [\text{new } A()]_{\text{priv}}; x\}|y \mapsto \langle \text{new } A(), p_- \rangle \text{ with (BLOCK-CTX) and} \\
&\quad y|y \mapsto \langle \text{new } A(), \text{pub} \rangle \xrightarrow{\text{priv}} \text{new } A()|y \mapsto \langle \text{new } A(), p_- \rangle \\
&\xrightarrow{\text{priv}} x|y \mapsto \langle \text{new } A(), p_- \rangle, x \mapsto \langle \text{new } A(), \text{priv} \rangle \text{ with (BLOCK)} \\
&\xrightarrow{\text{priv}} \text{new } A()|y \mapsto \langle \text{new } A(), p_- \rangle, x \mapsto \langle \text{new } A(), \text{priv} \rangle \text{ with (VAR)}
\end{aligned}$$

Instead reduction with grade `public` would be stuck since `pub` $\not\preceq$ `priv` and so

$$x|y \mapsto \langle \text{new } A(), p_- \rangle, x \mapsto \langle \text{new } A(), \text{priv} \rangle \not\rightarrow_{\text{pub}}$$

Also the reduction of $e_2 = \{A\ y = [\text{new } A()]_{\text{priv}}; \{A\ x = [y]_{\text{pub}}; x\}\}$ with grade `private`

$$\begin{aligned}
e_2|\emptyset &\xrightarrow{\text{priv}} \{A\ x = [y]_{\text{pub}}; x\}|y \mapsto \langle \text{new } A(), \text{priv} \rangle \text{ with (BLOCK)} \\
&\not\rightarrow_{\text{priv}}
\end{aligned}$$

would be stuck since $y|y \mapsto \langle \text{new } A(), \text{priv} \rangle \not\rightarrow_{\text{pub}}$. Note that both e_1 and e_2 reduce to `new A()` with the semantics of Figure 1.

⁴ Such kind of applications are called *informational* in [1].

⁵ As in *viewpoint adaptation* [13], where permission to a field access can be restricted based on the permission to the base object.

2. Let $e_3 = \{A\ x = [\text{new } A()]_{\text{pub}}; \text{new Pair}([x]_{\text{pub}}, [x]_{\text{priv}})\}$, e_3 starting with the empty environment reduces with grade **public** as follows:

$$\begin{aligned}
e_3|\emptyset &\rightarrow_{\text{pub}} \text{new Pair}([x]_{\text{pub}}, [x]_{\text{priv}})|x \mapsto \langle \text{new } A(), \text{pub} \rangle \text{ with (BLOCK)} \\
&\rightarrow_{\text{pub}} \text{new Pair}([\text{new } A()]_{\text{pub}}, [x]_{\text{priv}})|x \mapsto \langle \text{new } A(), \text{p}_- \rangle \text{ with (NEW-CTX) and} \\
&\quad x|x \mapsto \langle \text{new } A(), \text{pub} \rangle \rightarrow_{\text{pub}} \text{new } A()|x \mapsto \langle \text{new } A(), \text{p}_- \rangle \\
&\rightarrow_{\text{pub}} \text{new Pair}([\text{new } A()]_{\text{pub}}, [\text{new } A()]_{\text{priv}})|x \mapsto \langle \text{new } A(), \text{p}_- \rangle \text{ with (NEW-CTX) and} \\
&\quad x|x \mapsto \langle \text{new } A(), \text{p}_- \rangle \rightarrow_{\text{priv}} \text{new } A()|x \mapsto \langle \text{new } A(), \text{p}_- \rangle
\end{aligned}$$

It is easy to see that also $e_3|\emptyset \rightarrow_{\text{priv}}^* \text{new Pair}([\text{new } A()]_{\text{pub}}, [\text{new } A()]_{\text{priv}})|x \mapsto \langle \text{new } A(), \text{p}_- \rangle$. So we have

$$[e_3]_r.f|\emptyset \rightarrow_s^* [\text{new Pair}([\text{new } A()]_{\text{pub}}, [\text{new } A()]_{\text{priv}})]_r.f|x \mapsto \langle \text{new } A(), \text{p}_- \rangle$$

where f can be either **first** or **second** and r and s can be either **pub** or **priv**. Now, the reductions of grade **priv** accessing either **first** or **second** produce the value of the fields

$$[\text{new Pair}([\text{new } A()]_{\text{pub}}, [\text{new } A()]_{\text{priv}})]_r.f|_ \rightarrow_{\text{priv}} \text{new } A()|_$$

However, looking at the reductions of grade **pub**, only

$$[\text{new Pair}([\text{new } A()]_{\text{pub}}, [\text{new } A()]_{\text{priv}})]_{\text{pub}}.\text{first}|_ \rightarrow_{\text{pub}} \text{new } A()|_$$

is not stuck. That is, we produce a value that can be used as **public** only if we get a **public** field of a **public** object, whereas any value can be used as **private**.

We now state some simple properties of the semantics we will use to prove type soundness. The former establishes that reduction does not remove variables from the environment, the latter states that we can always decrease the grade of a reduction step.

► **Proposition 7.** *If $e|\rho \rightarrow_r e'|\rho'$ then $\text{dom}(\rho) \subseteq \text{dom}(\rho')$ and for all $x \in \text{dom}(\rho)$, $\rho(x) = \langle v, r \rangle$ implies $\rho'(x) = \langle v, s \rangle$ with $s \preceq r$.*

► **Proposition 8.** *If $e|\rho \rightarrow_r e'|\rho'$ and $s \preceq r$ then $e|\rho \rightarrow_s e'|\rho'$.*

We expect the instrumented reduction to be *sound* with respect to the standard reduction, in the sense that by erasing annotations from an instrumented reduction sequence we get a standard reduction sequence. This is formally stated below.

For any e expression, let us denote by $[e]$ the expression obtained by erasing annotations, defined in the obvious way, and analogously for environments, where grades associated to variables are removed as well.

► **Proposition 9** (Soundness of instrumented semantics).

If $e|\rho \rightarrow_r e'|\rho'$, then $[e]||[\rho] \rightarrow [e']||[\rho']$.

The converse does not hold, since a configuration could be annotated in a way that makes it stuck; notably, some resource (variable) could be exhausted or some field of an object could not be extracted. The graded type system in the next section will generate annotations which ensure soundness, hence also completeness with respect to the standard reduction.

4 Graded Featherweight Java

Types (class names) are annotated with *grades*, as shown in Figure 3.

As anticipated at the end of Section 2, a *coefficient context*, of shape $\gamma = x_1 : r_1, \dots, x_n : r_n$, where order is immaterial and $x_i \neq x_j$ for $i \neq j$, represents a map from variables to grades (called *coeffects* when used in this position) where only a finite number of variables have

3:12 Multi-Graded Featherweight Java

e	$::= x \mid e.f \mid \mathbf{new} C(es) \mid e.m(es) \mid \{T x = e; e'\}$	expression
T	$::= C^r$	(graded) type
v	$::= \mathbf{new} C(vs)$	value

■ **Figure 3** Syntax with grades.

non-zero coeffect. A (*type-and-coeffect*) *context*, of shape $\Gamma = x_1 :_{r_1} C_1, \dots, x_n :_{r_n} C_n$, with analogous conventions, represents the pair of the standard type context $x_1 : C_1 \dots, x_n : C_n$, and the coeffect context $x_1 : r_1, \dots, x_n : r_n$. We write $\mathbf{dom}(\Gamma)$ for $\{x_1, \dots, x_n\}$.

As customary in type-and-coeffect systems, in typing rules contexts are combined by means of some operations, which are, in turn, defined in terms of the corresponding operations on coeffects (grades). More precisely, we define:

- a partial order \preceq

$$\begin{aligned} \emptyset &\preceq \emptyset \\ x :_s C, \Gamma &\preceq x :_r C, \Delta && \text{if } s \preceq r \text{ and } \Gamma \preceq \Delta \\ \Gamma &\preceq x :_r C, \Delta && \text{if } x \notin \mathbf{dom}(\Gamma) \text{ and } \Gamma \preceq \Delta \end{aligned}$$

- a sum $+$

$$\begin{aligned} \emptyset + \Gamma &= \Gamma \\ (x :_s C, \Gamma) + (x :_r C, \Delta) &= x :_{s+r} C, (\Gamma + \Delta) \\ (x :_s C, \Gamma) + \Delta &= x :_s C, (\Gamma + \Delta) && \text{if } x \notin \mathbf{dom}(\Delta) \end{aligned}$$

- a scalar multiplication \cdot

$$s \cdot \emptyset = \emptyset \qquad s \cdot (x :_r C, \Gamma) = x :_{s \cdot r} C, (s \cdot \Gamma)$$

As the reader may notice, these operations on type-and-coeffect contexts can be equivalently defined by lifting the corresponding operations on coeffect contexts, which are the pointwise extension of those on coeffects, to handle types as well. In this step, the addition becomes partial since a variable in the domain of both contexts is required to have the same type.

The type system relies on the type information extracted from the class table, which, again to be concise, is abstractly modeled as follows:

- the subtyping relation \leq on class names is the reflexive and transitive closure of the **extends** relation
- $\mathbf{mtype}(C, m)$ gives, for each method m of class C , its enriched method type, where the types of the parameters and of **this** have coeffect annotations.

Moreover, $\mathbf{fields}(C)$ gives now a sequence $C_1^{r_1} f_1; \dots, C_n^{r_n} f_n;$, meaning that, to construct an object of type C , we need to provide, for each $i \in 1..n$, a value with a grade at least r_i .

The subtyping relation on graded types is defined as follows:

$$C^r \leq D^s \text{ iff } C \leq D \text{ and } s \preceq r$$

That is, a graded type is a subtype of another if the class is a heir class and the grade is more constraining. For instance, taking the affinity grade algebra of Example 2(2), an invocation of a method with return type C^ω can be used in a context where a type C^1 is required, e.g., to initialize a C^1 variable.

The typing judgment has shape $\Gamma \vdash e : T \rightsquigarrow e'$, where Γ is a type-and-coeffect context, and e' is an annotated expression, as defined in Figure 2. That is, typechecking generates annotations in code such that evaluation cannot get stuck, as will be formally expressed and proved in the following.

$$\begin{array}{c}
\text{(T-SUB)} \quad \frac{\Gamma \vdash e : T \rightsquigarrow e' \quad \Gamma \preceq \Gamma'}{\Gamma' \vdash e : T' \rightsquigarrow e' \quad T \leq T'} \quad \text{(T-VAR)} \quad \frac{}{x :_r C \vdash x : C^r \rightsquigarrow x} \quad r \neq \mathbf{0} \\
\text{(T-FIELD-ACCESS)} \quad \frac{\Gamma \vdash e : C^r \rightsquigarrow e'}{\Gamma \vdash e.f_i : C_i^{r \cdot r_i} \rightsquigarrow [e']_{r \cdot r_i}} \quad \text{fields}(C) = C_1^{r_1} f_1; \dots C_n^{r_n} f_n; \\
\text{(T-NEW)} \quad \frac{\Gamma_i \vdash e_i : C_i^{r \cdot r_i} \rightsquigarrow e'_i \quad \forall i \in 1..n}{\Gamma_1 + \dots + \Gamma_n \vdash \mathbf{new} C(e_1, \dots, e_n) : C^r \rightsquigarrow \mathbf{new} C([e'_1]_{r_1}, \dots, [e'_n]_{r_n})} \quad \text{fields}(C) = C_1^{r_1} f_1; \dots C_n^{r_n} f_n; \\
\text{(T-INVK)} \quad \frac{\Gamma_0 \vdash e_0 : C^{r_0} \rightsquigarrow e'_0 \quad \Gamma_i \vdash e_i : C_i^{r_i} \rightsquigarrow e'_i \quad \forall i \in 1..n}{\Gamma_0 + \dots + \Gamma_n \vdash e_0.m(e_1, \dots, e_n) : T \rightsquigarrow [e'_0]_{r_0}.m([e'_1]_{r_1}, \dots, [e'_n]_{r_n})} \quad \text{mtype}(C, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \rightarrow T \\
\text{(T-BLOCK)} \quad \frac{\Gamma_1 \vdash e_1 : C^r \rightsquigarrow e'_1 \quad \Gamma_2, x :_r C \vdash e_2 : T \rightsquigarrow e'_2}{\Gamma_1 + \Gamma_2 \vdash \{C^r x = e_1; e_2\} : T \rightsquigarrow \{C x = [e'_1]_r; e'_2\}} \\
\text{(T-ENV)} \quad \frac{\vdash v_i : C_i^{r_i} \rightsquigarrow v'_i \quad \forall i \in 1..n}{\Gamma \vdash \rho \rightsquigarrow \rho'} \quad \begin{array}{l} \Gamma = x_1 :_{r_1} C_1, \dots, x_n :_{r_n} C_n \\ \rho = x_1 \mapsto \langle v_1, r_1 \rangle, \dots, x_n \mapsto \langle v_n, r_n \rangle \\ \rho' = x_1 \mapsto \langle v'_1, r_1 \rangle, \dots, x_n \mapsto \langle v'_n, r_n \rangle \end{array} \\
\text{(T-CONF)} \quad \frac{\Delta \vdash e : T \rightsquigarrow e' \quad \Gamma \vdash \rho \rightsquigarrow \rho'}{\Gamma \vdash e|\rho : T \rightsquigarrow e'|\rho'} \quad \Delta \preceq \Gamma
\end{array}$$

■ **Figure 4** Graded type system.

In a well-typed class table, method bodies are expected to conform to method types. That is, $\text{mtype}(C, m)$ and $\text{mbody}(C, m)$ should be either both undefined or both defined with the same number of parameters. In the latter case, the method body should be well-typed with respect to the method type, notably by typechecking the method body we should get coeffects which are (overapproximated by) those specified in the annotations. Formally, if $\text{mbody}(C, m) = \langle x_1 \dots x_n, e \rangle$, and $\text{mtype}(C, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \rightarrow T$, then the following condition must hold:

$$\text{(T-METH)} \quad \mathbf{this} :_{r_0} C, x_1 :_{r_1} C_1, \dots, x_n :_{r_n} C_n \vdash e : T \rightsquigarrow e'$$

Moreover, we assume the standard coherence conditions on the class table with respect to inheritance. That is, if $C \leq D$, then $\text{fields}(D)$ is a prefix of $\text{fields}(C)$ and, if $\text{mtype}(C, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \rightarrow T$, then $\text{mtype}(D, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \rightarrow T'$ with $T' \leq T$.

In Figure 4, we describe the typing rules, which are *parameterized* on the underlying grade algebra.

In rule (T-SUB), both the coeffect context and the (graded) type can be made more general. This means that, on one hand, variables can get less constraining coeffects. For instance, assuming again affinity coeffects, an expression which can be typechecked assuming to use a given variable at most once (coeffect 1) can be typechecked as well with no constraints (coeffect ω). On the other hand, recalling that grades are contravariant in types, an expression can get a more constraining grade. For instance, an expression of grade ω can be used where a grade 1 is required.

If we take $r = \mathbf{1}$, then rule (T-VAR) is analogous to the standard rule for variable in coeffect systems, where the coeffect context is the map where the given variable is used once, and no other is used. Here, more generally, the variable can get an arbitrary grade r , provided that it gets the same grade in the context. However, the use of the variable cannot be just discarded, as expressed by the side condition $r \neq \mathbf{0}$.

In rule (T-FIELD-ACCESS), the grade of the field is multiplied by the grade of the receiver. As already mentioned, this is a form of *viewpoint adaptation* [13]. For instance, using affinity grades, a field graded ω of an object graded 1 can be used at most once.

3:14 Multi-Graded Featherweight Java

In rule (T-NEW), analogously to rule (T-VAR), the constructor invocation can get an arbitrary grade r , provided that the grades of the fields are multiplied by the same grade. Coeffects of the subterms are summed, as customary in type-and-coeffect systems.

In rule (T-INVK), the coeffects of the arguments are summed as well. The rule uses the function `mtype` on the class table, which, given a class name and a method name, returns its parameter and return (graded) types. For the implicit parameter `this` only the grade is specified. Note that the grades of the parameters are used in two different ways: as (part of) types, when typechecking the arguments; as coeffects, when typechecking the method body.

In rule (T-BLOCK), the coeffects of the initialization expression are summed with those of the body, excluding the local variable. Analogously to method parameters, the grade of the local variable is both used as (part of) type, when typechecking the initialization expression, and as coeffect, when typechecking the body.

Finally, we have straightforward rules for typing environments and configurations. Values in the environment are assumed to be closed, since we are in a call-by-value calculus. Also note that, in the judgment for environments and configurations, since no subsumption rule is available, variables in the context are exactly those in the domain of the environment, which are a superset of those used in the expression.

► **Example 10.** We show a simple example illustrating the use of graded types, assuming affinity grades. We write in square brackets the grade of the implicit `this` parameter. The class `Pair` declares three versions of the getter for the `first` field, which differ for the grade of the result: either 0, meaning that the result of the method *cannot* be used, or 1, meaning it can be used at most once, or ω , meaning it can be used with no constraints. Note that the first version, clearly useless in a functional calculus, could make sense adding effects, e.g. in an imperative calculus, playing a role similar to that of `void`.

```
class Pair { A1 first; A1 second;
  A0 getFirstZero() [1]{this.first}
  A1 getFirstAffine() [1]{this.first}
  Aω getFirst() [1]{this.first}
}
```

The coeffect of `this` is 1 in all versions, and it is actually used once in the bodies. The occurrence of `this` in the bodies can get any non-zero grade thanks to rule (T-VAR), and fields are graded 1, meaning that a field access does not affect the grade of the receiver, hence the three bodies can get any non-zero grade as well, so they are well-typed with respect to the grade in the method return type.

In the client code below, a call of the getter is assigned to a local variable of the same grade, which is then used consistently with such grade.

```
Pair1 p = ...
{A0 a = p.getFirstZero(); new Pair(new A(),new A())}
{A1 a = p.getFirstAffine(); new Pair(a,new A())}
{Aω a = p.getFirst(); new Pair(a,a)}
```

The following blocks are, instead, ill-typed, for two different reasons.

```
{A1 a = p.getFirst(); new Pair(a,a)}
{Aω a = p.getFirstAffine(); new Pair(a,a)}
```

In the first one, the initialization is correct, by subsumption, since we use an expression of a less constrained grade. However, the variable is then used in a way which is not compatible with its grade. In the second one, instead, the variable is used consistently with its grade, but the initialization is ill-typed, since we use an expression of a more constrained grade.

Finally, note that the coefficient of `this` could be safely changed to be ω in the three methods, providing an overapproximated information; in this case, however, the three invocations in the client code would be wrong, since the receiver `p` is required to be used at most once.

► **Example 11.** Consider the following source (that is, non-annotated) version of the expression in Example 5.

```
{Apublic y = new A(); {Aprivate x = y; x}}
```

The `private` variable `x` is initialized with the `public` expression/variable `y`. The block expression has type A^{private} as the following type derivation shows.

$$\text{(T-BLOCK)} \frac{\text{(T-NEW)} \frac{}{\vdash \text{new A}() : A^{\text{pub}}} \mathcal{D}}{\vdash \{A^{\text{pub}} y = \text{new A}(); \{A^{\text{priv}} x = y; x\}\} : A^{\text{priv}}}$$

where \mathcal{D} is the following derivation

$$\text{(T-BLOCK)} \frac{\text{(T-SUB)} \frac{\text{(T-VAR)} \frac{}{y :_{\text{pub}} A \vdash y : A^{\text{pub}}}}{y :_{\text{pub}} A \vdash y : A^{\text{priv}}} \quad \text{(T-VAR)} \frac{}{y :_{\text{pub}} A, x :_{\text{priv}} A \vdash x : A^{\text{priv}}}}{y :_{\text{pub}} A \vdash \{A^{\text{priv}} x = y; x\} : A^{\text{priv}}}}$$

On the other hand, initializing a `public` variable with a `private` expression as in

```
{Aprivate y = new A(); {Apublic x = y; x}}
```

is not possible, as expected, since $y :_{\text{priv}} A \not\vdash y : A^{\text{pub}}$.

Consider now the class `Pair` with a `private` field and a `public` one.

```
class B { Apublic f1; Aprivate f2; }
```

The expression e

```
{Apublic x = new A(); new B(x, x)}
```

can be given type $\text{Pair}^{\text{public}}$ as follows:

$$\text{(T-BLOCK)} \frac{\text{(T-NEW)} \frac{}{\vdash \text{new A}() : A^{\text{pub}}} \quad \text{(T-NEW)} \frac{\text{(T-VAR)} \frac{\text{(T-VAR)} \frac{}{x :_{\text{pub}} A \vdash x : A^{\text{pub}}} \quad \text{(T-SUB)} \frac{}{x :_{\text{pub}} A \vdash x : A^{\text{priv}}}}{x :_{\text{pub}} A \vdash \text{new Pair}(x, x) : \text{Pair}^{\text{pub}}}}{x :_{\text{pub}} A \vdash \text{new Pair}(x, x) : \text{Pair}^{\text{pub}}}}{\vdash \{A^{\text{pub}} x = \text{new A}(); \text{new Pair}(x, x)\} : \text{Pair}^{\text{pub}}}}$$

By (T-SUB) we can also derive $\vdash e : \text{Pair}^{\text{priv}}$ and so we get

$$\text{(T-FIELD)} \frac{\vdash e : \text{Pair}^{\text{priv}}}{\vdash e.\text{first} : A^{\text{priv}}} \quad \text{(T-FIELD)} \frac{\vdash e : \text{Pair}^{\text{pub}}}{\vdash e.\text{second} : A^{\text{priv}}}$$

that is, accessing a `public` field of a `private` expression we get a `private` result as well as accessing a `private` field of a `public` expression.

Also note that the following expression e'

```
{Aprivate x = new A(); new B(x, x)}
```

can be given only type $\text{Pair}^{\text{private}}$ by

$$\text{(T-BLOCK)} \frac{\text{(T-NEW)} \frac{}{\vdash \text{new A}() : A^{\text{priv}}} \quad \text{(T-NEW)} \frac{\text{(T-VAR)} \frac{}{x :_{\text{priv}} A \vdash x : A^{\text{priv}}} \quad \text{(T-VAR)} \frac{}{x :_{\text{priv}} A \vdash x : A^{\text{priv}}}}{x :_{\text{priv}} A \vdash \text{new Pair}(x, x) : \text{Pair}^{\text{priv}}}}{x :_{\text{priv}} A \vdash \text{new Pair}(x, x) : \text{Pair}^{\text{priv}}}}$$

We cannot derive $\vdash e' : \text{Pair}^{\text{pub}}$, since the grade of `first` is `public` and (T-NEW) would require $x :_{\text{priv}} A \vdash x : A^{\text{pub}\cdot\text{pub}}$, which does not hold.

3:16 Multi-Graded Featherweight Java

$$\begin{array}{c}
\text{(T-SUB)} \quad \frac{\Gamma \vdash_a e : T \quad \Gamma \preceq \Gamma'}{\Gamma' \vdash_a e : T'} \quad T \leq T' \quad \text{(T-VAR)} \quad \frac{}{x :_r C \vdash_a x : C^r} \quad r \neq \mathbf{0} \\
\text{(T-FIELD-ACCESS)} \quad \frac{\Gamma \vdash_a e : C^r}{\Gamma \vdash_a [e]_r . f_i : C_i^{r \cdot r_i}} \quad \text{fields}(C) = C_1^{r_1} f_1 ; \dots C_n^{r_n} f_n ; \\
\text{(T-NEW)} \quad \frac{\Gamma_i \vdash_a e_i : C_i^{r \cdot r_i} \quad \forall i \in 1..n}{\Gamma_1 + \dots + \Gamma_n \vdash_a \text{new } C([e_1]_{r_1}, \dots, [e_n]_{r_n}) : C^r} \quad \text{fields}(C) = C_1^{r_1} f_1 ; \dots C_n^{r_n} f_n ; \\
\text{(T-INVK)} \quad \frac{\Gamma_0 \vdash_a e_0 : C^{r_0} \quad \Gamma_i \vdash_a e_i : C_i^{r_i} \quad \forall i \in 1..n}{\Gamma_0 + \dots + \Gamma_n \vdash_a [e_0]_{r_0} . m([e_1]_{r_1}, \dots, [e_n]_{r_n}) : T} \quad \text{mtype}(C, m) = r_0, C_1^{r_1} \dots C_n^{r_n} \rightarrow T \\
\text{(T-BLOCK)} \quad \frac{\Gamma_1 \vdash_a e_1 : C^r \quad \Gamma_2, x :_r C \vdash_a e_2 : T}{\Gamma_1 + \Gamma_2 \vdash_a \{ C x = [e_1]_r ; e_2 \} : T} \\
\text{(T-ENV)} \quad \frac{\vdash_a v_i : C_i^{r_i} \quad \forall i \in 1..n \quad \Gamma = x_1 :_{r_1} C_1, \dots, x_n :_{r_n} C_n}{\Gamma \vdash_a \rho} \quad \rho = x_1 \mapsto \langle v_1, r_1 \rangle, \dots, x_n \mapsto \langle v_n, r_n \rangle \\
\text{(T-CONF)} \quad \frac{\Delta \vdash_a e : T \quad \Gamma \vdash_a \rho}{\Gamma \vdash_a e | \rho : T} \quad \Delta \preceq \Gamma
\end{array}$$

■ **Figure 5** Graded type system for annotated syntax.

Resource-aware soundness. We state that the graded type system is sound with respect to the resource-aware semantics. In other words, the graded type system prevents both standard typing errors, such as invoking a missing field or method, and resource-usage errors, such as requiring a resource which is exhausted (cannot be used in the needed way).

In order to state and prove a soundness theorem, we need to introduce a (straightforward) typing judgment \vdash_a for annotated expressions, environments and configurations. The typing rules are reported in Figure 5.

Recall that $[_]$ denotes erasing annotations. It is easy to see that an annotated expression is well-typed if and only if it is produced by the type system:

► **Proposition 12.** $\Gamma \vdash e : T \rightsquigarrow e'$ if and only if $[e'] = e$ and $\Gamma \vdash_a e' : T$.

A similar property holds for environments and configurations.

The main result is the following resource-aware progress theorem.

► **Theorem 13 (Resource-aware progress).** If $\Gamma \vdash_a e | \rho : C^r$ then either e is a value or $e | \rho \rightarrow_r e' | \rho'$ and $\Gamma' \vdash_a e' | \rho' : C^r$ with $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$ and $\Gamma' \preceq \Gamma, \Delta$.

When reduction is non-deterministic, we can distinguish two flavours of soundness, *soundness-must* meaning that no computation can be stuck, and *soundness-may*, meaning that at least one computation is not stuck. The terminology of *may* and *must* properties is very general and comes originally from [12]; the specific names *soundness-may* and *soundness-must* were introduced in [10, 9] in the context of big-step semantics. In our case, graded reduction is non-deterministic since, as discussed before, the rule (VAR) could be instantiated in different ways, possibly consuming the resource more than necessary. However, we expect that, for a well-typed configuration, there is at least one computation which is not stuck, hence a *soundness-may* result. *Soundness-may* can be proved by a theorem like the one above, which can be seen as a *subject-reduction-may* result, including standard progress. In our case, if the configuration is well-typed, that is, annotations have been generated by the type system, *there is a step* which leads, in turn, to a well-typed configuration. More in detail, the type is preserved, resources initially available may have reduced grades, and other available resources may be added.

Theorem 13 is proved as a special case of the following more general result, which makes explicit the invariant needed to carry out the induction. Indeed, by looking at the reduction rules, we can see that computational ones either add new variables to the environment or reduce the grade of a variable of some amount that depends on the grade of the reduction. In the latter case, the amount can be arbitrarily chosen with the only restrictions that it is non zero and at least the grade of the reduction. However, to prove progress, we not only have to prove that a reduction can be done, but, if the reduction is done in a context, say evaluating the argument of a constructor, then after such reduction we still have enough resources to go on with the reduction, that is, to evaluate the rest of the context (the other arguments of the constructor). This means that the resulting environment has enough resources to type the whole context (the constructor call). For this reason, in the statement of the theorem that follows, we add to the assumption of Theorem 13 a typing context Θ that would contain the information on the amount of resources that we want to preserve during the reduction (see Item 4 of the theorem). This allows us to choose the appropriate grade to be kept when reducing a variable and to reconstruct a typing derivation when using contextual reduction rules. For the expression at the top level, as we see from the proof of Theorem 13, Θ is simply $\mathbf{0}$ for all variables in the typing context in which the expression is typed.

► **Theorem 14.** *If $\Delta \vdash_a e : C^r$ and $\Gamma \vdash_a \rho$ and $\Delta + \Theta \preceq \Gamma$ and $\text{dom}(\Delta) \subseteq \text{dom}(\Theta)$ and e is not a value, then there are $e', \rho', \Delta', \Gamma'$ and Θ' such that*

1. $e|\rho \rightarrow_r e'|\rho'$ and
2. $\Delta' \vdash_a e' : C^r$ with $\Delta' \preceq \Delta, \Theta'$ and
3. $\Gamma' \vdash_a \rho'$ with $\Gamma' \preceq \Gamma, \Theta'$ and
4. $\Delta' + \Theta' \preceq \Gamma'$.

Finally, the following corollary states both subject-reduction for the standard semantics, that is, type and coeffects are preserved, and completeness of the instrumented semantics, that is, for well-typed configurations, every reduction step in the usual semantics can be simulated by an appropriate step in the instrumented semantics.

► **Corollary 15 (Subject reduction).** *If $\Gamma_1 \vdash e_1|\rho_1 : C^r \rightsquigarrow e'_1|\rho'_1$ and $e_1|\rho_1 \rightarrow e_2|\rho_2$, then $\Gamma_2 \vdash e_2|\rho_2 : C^r \rightsquigarrow e'_2|\rho'_2$ with $\text{dom}(\Gamma_1) \subseteq \text{dom}(\Gamma_2)$ and $\Gamma_2 \preceq \Gamma_1, \Delta$, and $e'_1|\rho'_1 \rightarrow_r e'_2|\rho'_2$.*

Proof. By Proposition 12 we get $\Gamma_1 \vdash_a e'_1|\rho'_1 : C^r$ and, by Theorem 13, $e'_1|\rho'_1 \rightarrow_r e'_2|\rho'_2$ and $\Gamma_2 \vdash_a e'_2|\rho'_2 : C^r$ with $\text{dom}(\Gamma_1) \subseteq \text{dom}(\Gamma_2)$ and $\Gamma_2 \preceq \Gamma_1, \Delta$. By Proposition 12, we get $\Gamma_2 \vdash [e'_2]|\rho'_2 : C^r \rightsquigarrow e_2|\rho_2$ and by Proposition 9, we get $e_1|\rho_1 \rightarrow [e'_2]|\rho'_2$. By the determinism of the standard semantics we have $[e'_2] = e_2$ and $[\rho'_2] = \rho_2$, hence the thesis. ◀

5 Combining grades

As we have seen, each grade algebra encodes a specific notion of resource usage. However, a program may need different notions of usage for different resources or different pieces of code (e.g., different classes). Hence, one needs to use several grade algebras at the same time, that is, a family $(H_k)_{k \in \mathcal{K}}$ of grade algebras⁶ indexed over a set \mathcal{K} of *grade kinds*. We assume grade kinds to always include \mathbf{N} and \mathbf{T} , with $H_{\mathbf{N}}$ and $H_{\mathbf{T}}$ the grade algebras of natural numbers and trivial, respectively, as in Example 2, since they play a special role, as will be shown.

⁶ H stands for “heterogeneous”.

- **Example 16.** Assume to use, in a program, grade kinds \mathbf{N} , \mathbf{A} , \mathbf{P} , \mathbf{PP} , \mathbf{AP} , and \mathbf{T} , where:
- $H_{\mathbf{A}}$ is the affinity grade algebra, as in Example 2(3).
 - $H_{\mathbf{P}}$ and $H_{\mathbf{PP}}$ are two different instantiations of the grade algebra of privacy levels, as in Example 6; namely, in $H_{\mathbf{P}}$ there are only two privacy levels `public` and `private`, whereas in $H_{\mathbf{PP}}$ we have privacy levels `a`, `b`, `c`, `d`, with $a \preceq b \preceq d$ and $a \preceq c \preceq d$.
 - Finally, $H_{\mathbf{AP}}$ is $H_{\mathbf{A}} \times H_{\mathbf{P}}$, as in Example 2(7), tracking simultaneously affinity and privacy.
- We want to make grades of all such kinds simultaneously available to the programmer. In order to achieve this, we should specify how to *combine* grades of different kinds through their distinctive operators; for instance, an object with grade of kind k could have a field with grade of kind μ , hence a field access should be graded by their multiplication.

In other words, we need to construct, starting from the family $(H_k)_{k \in \mathcal{K}}$, a single grade algebra of *heterogeneous grades*. In this way, the meta-theory developed in previous sections for an arbitrary grade algebra applies also to the case when several grade algebras are used at the same time. Note that this construction is necessary since we do not want available grades to be fixed, as in [23]; rather, the programmer should be allowed to define grades for a specific application, using some linguistic support which could be the language itself, as will be described in Section 6.

Direct refinement. The obvious approach is to define heterogeneous grades as pairs $\langle k, r \rangle$ where $k \in \mathcal{K}$, and $r \in H_k$. Concerning operators, in previous work, handling coefficients rather than grades, [3] we took the simplest choice, that is, combining (by either sum or product) grades of different kinds always returns $\langle \infty, \mathbf{T} \rangle$, meaning, in a sense, that we “do not know” how the combination should be done. The only exception are grades of kind \mathbf{N} ; indeed, since the corresponding grade algebra is initial, we know that, for any kind k , there is a unique grade homomorphism ι_k from \mathbf{Nat} to H_k , hence, to combine $\langle n, \mathbf{N} \rangle$ with $\langle r, k \rangle$, we can map n into a grade of kind k through such homomorphism, and then use the operator of kind k . In this paper, we generalize this idea, by allowing the programmer to specify, for each pair of kinds k and μ , a uniquely determined kind $k \oplus \mu$ and two uniquely determined grade homomorphisms $\text{lh}_{\kappa, \mu}^H : H_{\kappa} \rightarrow H_{\kappa \oplus \mu}$, and $\text{rh}_{\kappa, \mu}^H : H_{\mu} \rightarrow H_{\kappa \oplus \mu}$. In this way, to combine $\langle \kappa, r \rangle$ and $\langle \mu, s \rangle$, we can map both in grades of kind $k \oplus \mu$, and then use the operator of kind $k \oplus \mu$.

The operator \oplus and the family of unique homomorphisms, one for each pair of kinds, can be specified by the programmer, in a minimal and easy to check way, by defining a (*direct*) *refinement relation* \sqsubset^1 , as defined below, and a family of grade homomorphisms $H_{\kappa, \mu} : H_{\kappa} \rightarrow H_{\mu}$, indexed over pairs $\kappa \sqsubset^1 \mu$.

Given a relation \Rightarrow on kinds, a *path* from k_0 to k_n is a sequence $k_0 \dots k_n$ such as $k_i \Rightarrow k_{i+1}$, for all $i \in 1..n - 1$. We say that μ is an *ancestor* of κ if there is a path from κ to μ .

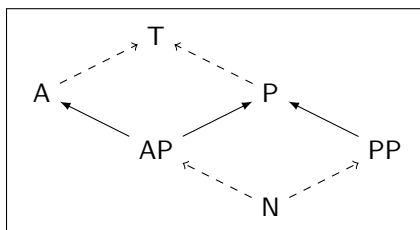
Then, a (*direct*) *refinement relation* is a relation \sqsubset^1 on $\mathcal{K} \setminus \{\mathbf{N}, \mathbf{T}\}$ such as the following conditions hold:

1. for each κ, μ , there exists at most one path from κ to μ
2. for each κ, μ with a common ancestor, there is a *least* common ancestor, denoted $\kappa \oplus \mu$; that is, such that, for any common ancestor ν , ν is an ancestor of $k \oplus \mu$.

Note that, thanks to requirement (1), requirement (2) means that the unique path, e.g., from κ to ν , consists of a unique path from κ to $k \oplus \mu$, and then a unique path from $k \oplus \mu$ to ν .

Given a direct refinement relation \sqsubset^1 , we can derive the following structure on \mathcal{K} :

- \sqsubset^1 can be extended to a partial order \sqsubseteq on \mathcal{K} , by taking the reflexive and transitive closure of \sqsubset^1 and adding $\mathbf{N} \sqsubseteq \kappa \sqsubseteq \mathbf{T}$ for all $\kappa \in \mathcal{K}$.
- \oplus can be extended to all pairs, by defining $\kappa \oplus \mu = \mathbf{T}$ if κ and μ have no common ancestor.



■ **Figure 6** Direct refinement diagram.

Altogether, we obtain an instance of a structure called *grade signature*, as will be detailed in Definition 18. Moreover, given a \sqsubset^1 -family of homomorphisms:

- they can be extended, by composition⁷, to all pairs of grades $\langle \kappa, \mu \rangle \in \mathcal{K} \setminus \{\mathbf{N}, \mathbf{T}\}$ such that there is a path from κ to μ ; since this path is unique, the resulting homomorphism is uniquely defined
- for each kind κ , we add the unique homomorphisms from \mathbf{Nat} and to \mathbf{Triv} .

Altogether, besides a grade algebra for each kind, we get a grade homomorphism for each pair $\langle \kappa, \mu \rangle$ such that $\kappa \sqsubseteq \mu$. That is, we obtain an instance of a structure called *heterogeneous grade algebra*, as will be detailed in Definition 19.

Thus, as desired, combining grades of kinds $\langle \kappa, r \rangle$ and $\langle \mu, s \rangle$ can be defined by mapping both r and s into grades of kind $\kappa \oplus \mu$, and then the operator of kind $\kappa \oplus \mu$ is applied.

The fact that in this way we actually obtain a grade algebra, that is, all required axioms are satisfied, is proved in the next subsection on the more general case of an arbitrary grade signature and heterogeneous grade algebra.

Note the special role played by the grade kinds \mathbf{N} and \mathbf{T} , with their corresponding grade algebras. The former turns out to be the minimal kind required in a grade signature (Definition 18); this is important since the zero and one of the resulting grade algebra (hence the zero and one used in the type system) will be those of this kind. The latter, as shown above, is used as default common ancestor for pairs of kinds which do not have one.

► **Example 17.** Coming back to our example, a programmer could define the direct refinement relation and the corresponding homomorphisms as follows:

- $\mathbf{PP} \sqsubset^1 \mathbf{P}$, and the homomorphism maps, e.g., \mathbf{a} , \mathbf{b} , and \mathbf{c} into **private** and \mathbf{d} into **public**
- $\mathbf{AP} \sqsubset^1 \mathbf{A}$, and $\mathbf{AP} \sqsubset^1 \mathbf{P}$, and the homomorphisms are the projections.

Thus, for instance, the grade $\langle \mathbf{AP}, \langle \omega, \mathbf{private} \rangle \rangle$, meaning that we can use the resource an arbitrary number of times in **private** mode, and $\langle \mathbf{PP}, \mathbf{d} \rangle$, meaning that we can use the resource in **d** mode, gives **private**. Indeed, both grades are mapped into the grade algebra of privacy levels $0 \preceq \mathbf{private} \preceq \mathbf{public}$; for the former, the information about the affinity is lost, whereas for the second the privacy level \mathbf{d} is mapped into **public**; finally, we get **private** = **private** · **public**.

The direct refinement relation is pictorially shown in Figure 6. Dotted arrows denote (some of) the order relations added for \mathbf{N} and \mathbf{T} .

Note that specifying the grade signature and the heterogeneous grade algebra indirectly, by means of the direct refinement relation and the corresponding homomorphisms, has a fundamental advantage: the semantic check that, for each κ, μ , we can map grades of grade κ into grades of kind μ in a unique way (that is, there is at most one homomorphism), which would require checking the equivalence of function definitions, is replaced by the checks (1)

⁷ Note that in this way we obtain, in particular, all the identities.

and (2) in the definition of direct refinement, which are purely syntactic and can be easily implemented in a type system (a simple stronger condition is to impose that each kind has a unique parent in the direct refinement relation, as it is for single inheritance).

In Section 6, we will see how to express both grade algebras and homomorphisms in Java; roughly, both will be represented by classes implementing a suitable generic interface.

A general construction. We provide a construction that, starting from a family of grade algebras with a suitable structure, yields a unique grade algebra summarising the whole family. As a consequence, the meta-theory developed in previous sections for a single grade algebra applies also to the case when several grade algebras are used at the same time.

To develop this construction, we use simple and standard categorical tools, referring to [20, 26] for more details. Given a category \mathcal{C} , we denote by \mathcal{C}_0 the collection of objects in \mathcal{C} and we say that \mathcal{C} is *small* when \mathcal{C}_0 is a set. Recall that any partially ordered set $\mathcal{P} = \langle \mathcal{P}_0, \sqsubseteq \rangle$ can be seen as a small category where objects are the elements of \mathcal{P}_0 and, for all $x, y \in \mathcal{P}_0$, there is an arrow $x \rightarrow y$ iff $x \sqsubseteq y$; hence, for every pair of objects in \mathcal{P}_0 , there is at most one arrow between them, and the only isomorphisms are the identities.

► **Definition 18.** A grade signature \mathcal{S} is a partially ordered set with finite suprema, that is, it consists of the following data:

- a partially ordered set $\langle \mathcal{S}_0, \sqsubseteq \rangle$;
- a function $\oplus: \mathcal{S}_0 \times \mathcal{S}_0 \rightarrow \mathcal{S}_0$ monotone in both arguments and such that for all $\kappa, \mu, \nu \in \mathcal{S}_0$, $\kappa \oplus \mu \sqsubseteq \nu$ iff $\kappa \sqsubseteq \nu$ and $\mu \sqsubseteq \nu$;
- a distinguished object $I \in \mathcal{S}_0$ such that $I \sqsubseteq \kappa$, for all $\kappa \in \mathcal{S}_0$.

Intuitively, objects in \mathcal{S} represent the *kinds* of grades one wants to work with, while the arrows, namely, the order relation, model a refinement between such kinds: $\kappa \sqsubseteq \mu$ means that the kind κ is *more specific* than the kind μ . The operation \oplus combines two kinds to produce the most specific kind generalising both. Finally, the kind I is the most specific one.

It is easy to check that the following properties hold for all $\kappa, \mu, \nu \in \mathcal{S}_0$:

$$\begin{aligned} (\kappa \oplus \mu) \oplus \nu &= \kappa \oplus (\mu \oplus \nu) & \kappa \oplus \kappa &= \kappa \\ \kappa \oplus \mu &= \mu \oplus \kappa & \kappa \oplus I &= \kappa \end{aligned}$$

namely, $\langle \mathcal{S}_0, \oplus, I \rangle$ is a commutative idempotent monoid.

► **Definition 19.** A heterogeneous grade algebra over the grade signature \mathcal{S} is just a functor $H: \mathcal{S} \rightarrow \mathit{GrAlg}$. That is, it consists of a grade algebra $H(\kappa)$, written also H_κ , for every kind $\kappa \in \mathcal{S}_0$, and a grade algebra homomorphism $H_{\kappa, \mu}: H_\kappa \rightarrow H_\mu$ for every arrow $\kappa \sqsubseteq \mu$, respecting composition and identities⁸, that is, $\kappa \sqsubseteq \mu \sqsubseteq \nu$ implies $H_{\kappa, \nu} = H_{\mu, \nu} \circ H_{\kappa, \mu}$ and $H_{\kappa, \kappa} = \text{id}_{H_\kappa}$.

Essentially, the homomorphisms $H_{\kappa, \mu}$ realise the refinement $\kappa \sqsubseteq \mu$, transforming grades of kind κ into grades of kind μ , preserving the grade algebra structure.

Observe that the arrows $I \sqsubseteq \kappa$ and $\kappa \sqsubseteq \kappa \oplus \mu$ and $\mu \sqsubseteq \kappa \oplus \mu$ in \mathcal{S} give rise to the following grade algebra homomorphisms:

$$\text{in}_\kappa^H = H_{I, \kappa}: H_I \rightarrow H_\kappa \quad \text{lh}_{\kappa, \mu}^H = H_{\kappa, \kappa \oplus \mu}: H_\kappa \rightarrow H_{\kappa \oplus \mu} \quad \text{rh}_{\kappa, \mu}^H = H_{\mu, \kappa \oplus \mu}: H_\mu \rightarrow H_{\kappa \oplus \mu}$$

⁸ The notation $H_{\kappa, \mu}$ makes sense, since between κ and μ there is at most one arrow.

which provide us with a way to map grades of kind I into grades of any other kind, and grades of kind κ and μ into grades of their composition $\kappa \oplus \mu$. By functoriality of H and using the commutative idempotent monoid structure of \mathcal{S} , we get the following equalities hold in the category \mathcal{GrAlg} , ensuring consistency of such transformations:

$$\text{lh}_{\kappa \oplus \mu, \nu}^H \circ \text{lh}_{\kappa, \mu}^H = \text{lh}_{\kappa, \mu \oplus \nu}^H \quad (1)$$

$$\text{rh}_{\kappa \oplus \mu, \nu}^H \circ \text{lh}_{\kappa, \mu}^H = \text{rh}_{\kappa, \mu \oplus \nu}^H \circ \text{lh}_{\mu, \nu}^H \quad (2)$$

$$\text{lh}_{\kappa, \mu}^H = \text{rh}_{\mu, \kappa}^H \quad (3)$$

$$\text{lh}_{\kappa, \kappa}^H = \text{id}_{H_\kappa} \quad (4)$$

$$\text{lh}_{\kappa, I}^H = \text{id}_{H_\kappa} \quad (5)$$

$$\text{rh}_{\kappa, I}^H = \text{in}_\kappa^H \quad (6)$$

In the following, we will show how to turn a heterogeneous grade algebra into a single grade algebra. The procedure we will describe is based on a general construction due to Grothendieck [17] defined on indexed categories.

Let us assume a grade signature \mathcal{S} and a heterogeneous grade algebra $H: \mathcal{S} \rightarrow \mathcal{GrAlg}$. We consider the following set:

$$|G(H)| = \{\langle \kappa, r \rangle \mid \kappa \in \mathcal{S}_0, r \in |H_\kappa|\}$$

That is, elements of $G(H)$ will be *kinded grades*, namely, pairs of a kind κ and a grade of that kind. Note that this is indeed a set because \mathcal{S} is small, that is, \mathcal{S}_0 is a set. Then, we define a binary relation \preceq_H on $|G(H)|$ as follows:

$$\langle \kappa, r \rangle \preceq_H \langle \mu, s \rangle \quad \text{iff} \quad \kappa \sqsubseteq \mu \quad \text{and} \quad H_{\kappa, \mu}(r) \preceq_\mu s$$

that is, the kind κ must be more specific than the kind μ and, transforming r by $H_{\kappa, \mu}$, we obtain a grade of kind μ which is smaller than s . These data define a partially ordered set as the following proposition shows.

► **Proposition 20.** $\langle |G(H)|, \preceq_H \rangle$ is a partially ordered set.

The additive structure is given by a binary operation $+_H: |G(H)| \times |G(H)| \rightarrow |G(H)|$ and an element $\mathbf{0}_H$ in $|G(H)|$ defined as follows:

$$\langle \kappa, r \rangle +_H \langle \mu, s \rangle = \langle \kappa \oplus \mu, \text{lh}_{\kappa, \mu}^H(r) +_{\kappa \oplus \mu} \text{rh}_{\kappa, \mu}^H(s) \rangle \quad \mathbf{0}_H = \langle I, \mathbf{0}_I \rangle$$

That is, the addition of $\langle \kappa, r \rangle$ and $\langle \mu, s \rangle$ is performed by first mapping r and s in the most specific kind generalising both κ and μ , namely $\kappa \oplus \mu$, and then by summing them in the grade algebra over that kind. The zero element is just the zero of the most specific kind.

► **Proposition 21.** $\langle |G(H)|, \preceq_H, +_H, \mathbf{0}_H \rangle$ is an ordered commutative monoid.

► **Proposition 22.** $\mathbf{0}_H \preceq_H \langle \kappa, r \rangle$ for every $\langle \kappa, r \rangle \in |G(H)|$.

Similarly, the multiplicative structure is given by a binary operation $\cdot_H: |G(H)| \times |G(H)| \rightarrow |G(H)|$ and an element $\mathbf{1}_H$ in $|G(H)|$ defined as follows:

$$\langle \kappa, r \rangle \cdot_H \langle \mu, s \rangle = \begin{cases} \langle \kappa \oplus \mu, \text{lh}_{\kappa, \mu}^H(r) \cdot_{\kappa \oplus \mu} \text{rh}_{\kappa, \mu}^H(s) \rangle & \langle \kappa, r \rangle \neq \mathbf{0}_H \quad \text{and} \quad \langle \mu, s \rangle \neq \mathbf{0}_H \\ \mathbf{0}_H & \text{otherwise} \end{cases} \quad \mathbf{1}_H = \langle I, \mathbf{1}_I \rangle$$

Notice that the definitions above follow almost the same pattern as additive operations, but we force that multiplying by $\mathbf{0}_H$ we get again $\mathbf{0}_H$, which is a key property of grade algebras.

► **Proposition 23.** $\langle |G(H)|, \preceq_H, \cdot_H, \mathbf{1}_H \rangle$ is an ordered monoid.

Altogether, we finally get the following result.

► **Theorem 24.** $G(H) = \langle |G(H)|, \preceq_H, +_H, \cdot_H, \mathbf{0}_H, \mathbf{1}_H \rangle$ is a grade algebra.

6 Grades as Java expressions

In Section 4 we described how a Java-like language could be equipped with *grades* decorating types, taken in an arbitrary grade algebra. Moreover, in Section 5, we have shown that such grade algebra could have been obtained by composing, in a specific way determined by providing a minimal collection of grade homomorphisms, different grade algebras corresponding to different ways to track usage of resources. In this section, we consider the issue of providing a linguistic support to this aim. This could be done by using an ad-hoc configuration language, however, we believe an interesting solution is that the grade annotations in types could be written themselves in Java.

The key idea is that grade annotations are Java expressions of (classes implementing) a predefined interface `Grade`, analogously to Java exceptions which are expressions of (subclasses of) `Exception`. Moreover, grade homomorphisms are user-defined as well. Namely, a user program can include:

- pairs of *grade classes* and *grade factory classes*, each one modeling a grade algebra desired for the specific application, with the factory class providing its constants
- *grade homomorphism classes*, each one modeling a homomorphism from a grade algebra (class) to another.

When typechecking code with grade annotations, the grades internally used by the type system are those obtained by combining all the declared grade algebras (classes) by means of the declared grade homomorphism classes, with the construction described in Section 5. Operations on grades in the same grade algebra (class) are derived from user-defined methods, as discussed more in detail below, whereas operations on heterogeneous grades are derived as in the construction in Section 5.

Grade and grade factory classes. They are classes implementing the following generic interfaces, respectively:

```
interface Grade<T extends Grade<T>> {
    boolean leq(T x);
    T sum(T x);
    T mult(T x);
}
interface GradeFactory<T extends Grade<T>>{
    T zero();
    T one();
}
```

Grade homomorphism classes. They are classes implementing the following generic interface:

```
interface GradeHom<T extends Grade<T>, R extends Grade<R>> {
    R apply(T x);
}
```

Examples of grade classes and grade homomorphism classes can be found in [4].

Typechecking could then be performed in two steps:

1. Code defining grades, which is assumed to be standard (that is, non-graded) Java code, is typechecked by the standard compiler.
2. Graded code (containing grade annotations written in Java) is typechecked accordingly to the graded type system in Figure 4, where the underlying grade algebra is obtained by composing, by the construction described in Section 5, the user-defined grade algebras through the user-defined grade homomorphisms. Each user-defined algebra has as carrier (set of grades) the Java values which are instances of the corresponding class, and the operations are computed by executing user-defined methods in such class. For instance, to compute the sum $v_1 + v_2$ of two grades which are values of a grade class, we evaluate $v_1.\text{sum}(v_2)$. Analogously to compute the result of a grade homomorphism.

For the whole process to work correctly, the following are responsibilities of the programmer:

- Grade classes, grade factory classes, and grade homomorphism classes should satisfy the axioms required for the structures they model, e.g., that the sum derived from `sum` methods is commutative and associative. The same happens, for instance, in Haskell, when one defines instances of `Functor` or `Monad`.
- Code defining grades should be *terminating*, since, as described above, the second typechecking step requires to *execute* code typechecked in the first step.
- Finally, the relation among grade classes implicitly defined by declaring grade homomorphism classes should actually be a direct refinement relation, that is, should satisfy the two requirements: (1) there exists at most one path between two grade classes, and (2) each two grade classes with a common ancestor have a *least* common ancestor. These are requirements easy to check, similarly to the check that inheritance is acyclic, or that there are no diamonds in multiple inheritance.

An interesting point is that implementations could use in a parametric way auxiliary tools, notably a termination checker to prevent divergence in methods implementing grade operations, and/or a verifier to ensure that they provide the required properties.

7 Related work

The two contributions which have been more inspiring for the work in this paper are the instrumented semantics proposed in [8] and the Granule language [23]. In [8], the authors develop GRAD, a graded dependent type system that includes functions, tensor products, additive sums, and a unit type. Moreover, they define an instrumented operational semantics which tracks usage of resources, and prove that the graded type system is sound with respect to such instrumented semantics. In this paper, we take the same approach to define a resource-aware semantics, parametric on an arbitrary grade algebra. However, differently from [8], where such semantics is defined on typed terms, with the only aim to show the role of the type system, the definition of our semantics is given *independently* from the type system, as is the standard approach in calculi. That is, the aim is also to provide a simple purely semantic model which takes into account usage of resources.

Granule [23] is a functional language equipped with graded modal types, where different kinds of grades can be used at the same time, including naturals for exact usage, security levels, intervals, infinity, and products of coeffects. We owe to Granule the idea of allowing different kinds of grades to coexist, and the overall objective to exploit graded modal types in a programming language. Concerning heterogeneous grades, in this paper we push forward the Granule approach, since we do not want this grade algebra to be fixed, but extendable

by the programmer with user-defined grades. To this aim we define the construction in Section 5. Concerning the design of a graded programming language, here we investigate the object-oriented rather than functional paradigm, taking some solutions which seem more adequate in that context, e.g., to have once-graded types and no boxing/unboxing. The design and implementation of a real Java-like language are not objectives of the current paper; however, we outline in Section 6 a possible interesting solution, where grade annotations are written in the language itself.

Coming more in general to resource-aware type systems, coeffects were first introduced by [24] and further analyzed by [25]. In particular, [25] develops a generic coeffect system which augments the simply-typed λ -calculus with context annotations indexed by *coeffect shapes*. The proposed framework is very abstract, and the authors focus only on two opposite instances: structural (per-variable) and flat (whole context) coeffects, identified by specific choices of context shapes.

Most of the subsequent literature on coeffects focuses on structural ones, for which there is a clear algebraic description in terms of semirings. This was first noticed by [7], who developed a framework for structural coeffects for a functional language. This approach is inspired by a generalization of the exponential modality of linear logic, see, e.g., [6]. That is, the distinction between linear and unrestricted variables of linear systems is generalized to have variables decorated by coeffects, or *grades*, that determine how much they can be used. In this setting, many advances have been made to combine coeffects with other programming features, such as computational effects [14, 23, 11], dependent types [2, 8, 22], and polymorphism [1]. Other graded type systems are explored in [2, 15, 1], also combining effects and coeffects [14, 23]. In all these papers, the process of tracking usage through grades is a powerful method of instrumenting type systems with analyses of irrelevance and linearity that have practical benefits like erasure of irrelevant terms (resulting in speed-up) and compiler optimizations (such as in-place update).

As already mentioned, [22] and [27] observed that contexts in a structural coeffect system form a module over the semiring of grades, even though they do not use this structure in its full generality, restricting themselves to free modules, that is, to structural coeffect systems. Recently, [5] shows a significant non-structural instance, namely, a coeffect system to track sharing in the imperative paradigm.

8 Conclusion

The contributions of the paper can be summarized as follows:

- Resource-aware extension of FJ reduction, parametric on an arbitrary grade algebra.
- Resource-aware extension of the type system, proved to ensure soundness-may of the resource-aware semantics.
- Formal construction which, given grades of different kinds and grade transformations corresponding to a refinement relation among kinds (formally, a functor over a grade signature), provides a grade algebra of *heterogeneous grades*.
- Notion of *direct refinement* allowing a minimal and easy to check way to specify the above functor.
- Outline of a Java extension where grades are user-defined, and grade annotations are written in the language itself.

As already noted, the key novel ideas in the contributions above are mostly independent from the language. So, a first natural direction for future work is to explore their incarnation in another paradigm, e.g., the functional one. That would include the definition of a parametric

resource-aware reduction independent from types, the design of a type system with once-graded types, and possibly the design of user-defined grades in a functional language, e.g., in Haskell by relying on the typeclass feature. Though the overall approach should still apply, we expect the investigation to be significant due to the specific features of the paradigm.

The resource-aware operational semantics defined in this paper requires *annotations* in subterms, with the only aim to fix their reduction grade in the reduction of the enclosing term. As mentioned in Section 3, adopting a big-step style would clearly remove the need of such technical artifice; only annotations in constructor subterms should be kept, since they express a true constraint on the semantics. Thus, a very interesting alternative to be studied is a big-step version of resource-aware semantics, allowing a more abstract and clean presentation. With this choice, we should employ, to prove soundness-may, the techniques recently introduced in [10, 9].

Coming back to Java-like languages, the FJ language considered in the paper does not include imperative features. Adding mutable memory leads to many significant research directions. First, besides the model presented in this paper, and in general in literature, where “using a resource” means “replacing a variable with its value”, another view is possible where the resource is the memory and “using” means “interacting with the memory”. Moreover, we would like to investigate more in detail how to express by grade algebras forms of usages which are typical of the imperative paradigm, such as the `readonly` modifier, and, more in general, *capabilities* [18, 16].

References

- 1 Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proceedings of ACM on Programming Languages*, 4(ICFP):90:1–90:28, 2020. doi:10.1145/3408972.
- 2 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *IEEE Symposium on Logic in Computer Science, LICS 2018*, pages 56–65. ACM Press, 2018. doi:10.1145/3209108.3209189.
- 3 Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. A Java-like calculus with user-defined coeffects. In Ugo Dal Lago and Daniele Gorla, editors, *ICTCS'22 – Italian Conference on Theoretical Computer Science*, volume 3284 of *CEUR Workshop Proceedings*, pages 66–78. CEUR-WS.org, 2022.
- 4 Riccardo Bianchini, Francesco Dagnino, Paola Giannini, and Elena Zucca. Multi-graded Featherweight Java. *CoRR*, 2023. URL: <http://arxiv.org/abs/2302.07782>.
- 5 Riccardo Bianchini, Francesco Dagnino, Paola Giannini, Elena Zucca, and Marco Servetto. Coeffects for sharing and mutation. *Proceedings of ACM on Programming Languages*, 6(OOPSLA):870–898, 2022. doi:10.1145/3563319.
- 6 Flavien Breuvert and Michele Pagani. Modelling coeffects in the relational semantics of linear logic. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic, CSL 2015*, volume 41 of *LIPICs*, pages 567–581. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPICs.CSL.2015.567.
- 7 Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. A core quantitative coeffect calculus. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2013*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer, 2014. doi:10.1007/978-3-642-54833-8_19.
- 8 Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proceedings of ACM on Programming Languages*, 5(POPL):1–32, 2021. doi:10.1145/3434331.
- 9 Francesco Dagnino. A meta-theory for big-step semantics. *ACM Transactions on Computational Logic*, 23(3):20:1–20:50, 2022. doi:10.1145/3522729.

- 10 Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In Peter Müller, editor, *European Symposium on Programming, ESOP 2020*, volume 12075 of *Lecture Notes in Computer Science*, pages 169–196. Springer, 2020. doi:10.1007/978-3-030-44914-8_7.
- 11 Ugo Dal Lago and Francesco Gavazzo. A relational theory of effects and coeffects. *Proceedings of ACM on Programming Languages*, 6(POPL):1–28, 2022. doi:10.1145/3498692.
- 12 Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1):83–133, 1984. doi:10.1016/0304-3975(84)90113-0.
- 13 Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In Erik Ernst, editor, *European Conference on Object-Oriented Programming, ECOOP 2007*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007.
- 14 Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvar, and Tarmo Uustalu. Combining effects and coeffects via grading. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *ACM International Conference on Functional Programming, ICFP 2016*, pages 476–489. ACM Press, 2016. doi:10.1145/2951913.2951939.
- 15 Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In Zhong Shao, editor, *European Symposium on Programming, ESOP 2013*, volume 8410 of *Lecture Notes in Computer Science*, pages 331–350. Springer, 2014. doi:10.1007/978-3-642-54833-8_18.
- 16 Colin S. Gordon. Designing with static capabilities and effects: Use, mention, and invariants (pearl). In Robert Hirschfeld and Tobias Pape, editors, *European Conference on Object-Oriented Programming, ECOOP 2020*, volume 166 of *LIPICs*, pages 10:1–10:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.ECOOP.2020.10.
- 17 Alexander Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental*, pages 145–194. Springer, 1971.
- 18 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In Theo D’Hondt, editor, *European Conference on Object-Oriented Programming, ECOOP 2010*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer, 2010.
- 19 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1999*, pages 132–146. ACM Press, 1999. doi:10.1145/320384.320395.
- 20 Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.
- 21 Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In Ilya Sergey, editor, *European Symposium on Programming, ESOP 2022*, volume 13240 of *Lecture Notes in Computer Science*, pages 346–375. Springer, 2022. doi:10.1007/978-3-030-99336-8_13.
- 22 Conor McBride. I got plenty o’ nuttin’. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.
- 23 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of ACM on Programming Languages*, 3(ICFP):110:1–110:30, 2019. doi:10.1145/3341714.
- 24 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages and Programming, ICALP 2013*, volume 7966 of *Lecture Notes in Computer Science*, pages 385–397. Springer, 2013. doi:10.1007/978-3-642-39212-2_35.
- 25 Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *ACM International Conference on Functional Programming, ICFP 2014*, pages 123–135. ACM Press, 2014. doi:10.1145/2628136.2628160.

- 26 Emily Riehl. *Category theory in context*. Courier Dover Publications, 2017.
- 27 James Wood and Robert Atkey. A framework for substructural type systems. In Ilya Sergey, editor, *European Symposium on Programming, ESOP 2022*, volume 13240 of *Lecture Notes in Computer Science*, pages 376–402. Springer, 2022. doi:10.1007/978-3-030-99336-8_14.




Hoogle \star : Constants and λ -abstractions in Petri-net-based Synthesis using Symbolic Execution

Henrique Botelho Guerra  

INESC-ID and IST, University of Lisbon, Portugal

João F. Ferreira   

INESC-ID and IST, University of Lisbon, Portugal

João Costa Seco   

NOVA LINCS, NOVA School of Science and Technology, Caparica, Portugal

Abstract

Type-directed component-based program synthesis is the task of automatically building a function with applications of available components and whose type matches a given goal type. Existing approaches to component-based synthesis, based on classical proof search, cannot deal with large sets of components. Recently, HOOGLE+, a component-based synthesizer for Haskell, overcomes this issue by reducing the search problem to a Petri-net reachability problem. However, HOOGLE+ cannot synthesize constants nor λ -abstractions, which limits the problems that it can solve.

We present HOOGLE \star , an extension to HOOGLE+ that brings constants and λ -abstractions to the search space, in two independent steps. First, we introduce the notion of *wildcard* component, a component that matches all types. This enables the algorithm to produce *incomplete functions*, i.e., functions containing occurrences of the wildcard component. Second, we complete those functions, by replacing each occurrence with constants or custom-defined λ -abstractions. We have chosen to find constants by means of an inference algorithm: we present a new unification algorithm based on symbolic execution that uses the input-output examples supplied by the user to compute substitutions for the occurrences of the wildcard.

When compared to HOOGLE+, HOOGLE \star can solve more kinds of problems, especially problems that require the generation of constants and λ -abstractions, without performance degradation.

2012 ACM Subject Classification Software and its engineering \rightarrow Automatic programming; Theory of computation \rightarrow Automated reasoning

Keywords and phrases Type-directed, component-based, program synthesis, symbolic execution, unification, Haskell

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.4

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.9.2.20>

Funding FCT UIDB/04516/2020, FCT UIDB/50021/2020, and ANI Lisboa-01-0247-Feder-045917.

Acknowledgements We want to thank to the anonymous reviewers, for the constructive feedback.

1 Introduction

Program synthesis is the task of automatically building a program that fulfills a specification supplied by the user [12]. Specifications can vary from examples [31], sketches [36], to ontologies [4] and types [13]. In type-guided component-based program synthesis, users provide the type of the function to synthesize (the *query type*), and optionally, input-output examples. Each solution is composed of applications of functions from a given *component set*. A recent example is HOOGLE+ [14, 20], a state-of-the-art synthesizer for the Haskell programming language that successfully solves several real-world problems with large component sets. For example, given the query type $(a \rightarrow b) \rightarrow [a] \rightarrow b$, it can



© Henrique Botelho Guerra, João F. Ferreira, and João Costa Seco;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 4; pp. 4:1–4:28



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



synthesize the function `\x1 x2 -> x1 (GHC.List.head x2)`. Multiple solutions are filtered using input-output examples. Unlike most approaches to component-based synthesis, which are based on classical proof search, HOOGLE+ can deal with large sets of components, because it reduces synthesis to a Petri-net reachability problem, following the approach of SYPET [7], a component-based synthesizer for Java.

Challenges for constants and λ -abstractions. Despite the benefits of Petri-net-based approaches, they exclude constants and custom λ -abstractions from the search space, because Petri nets only synthesize solutions whose terms belong to the component set, and it is impossible to insert all constants and custom λ -abstractions in a finite set. We found several problems in StackOverflow that HOOGLE+ cannot solve because the solutions require constants or λ -abstractions to be synthesized. So, bringing both classes of terms to the search space will allow HOOGLE+ to solve more problems, making life easier for Haskell programmers.

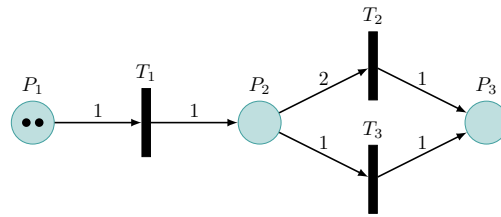
Motivating example. As an example, suppose that we want to append the constant 0 to a list. We provide to HOOGLE+ the query type `[Int] -> [Int]` together with the example that maps the input `[1]`¹ to the output `[1, 0]`. A solution is as simple as `\x1 -> x1 ++ [0]`, however, HOOGLE+ is not able to synthesize it, because it requires the constant `[0]` to be synthesized. The same happens with custom λ -abstractions. Suppose that we want to map each element of a list of integers to its square. For example, given `[1, 2, 3]`, the output should be `[1, 4, 9]`. The query type is `[Int] -> [Int]`, and a solution is `\list -> map (\x -> x * x) list`. However, this solution cannot be synthesized by HOOGLE+ as λ -abstractions do not belong to the search space.

Our approach. In this work we propose and evaluate a solution to bring constants and λ -abstractions to the search space of HOOGLE+, following two independent steps. First, we add to the component set the *wildcard component*, a component that matches all types. The Petri net is then allowed to synthesize *incomplete functions*: functions that use that component, such as `\list -> map wildcard list`. In this example, the wildcard component appears where a function is expected; however, in general, it could appear in place of an integer, string, or any other type. The second step is to replace the occurrences of the wildcard component with constants or λ -abstractions. When the wildcard occurs in place of a constant, we use a unification algorithm based on symbolic execution, that uses the input-output examples provided by the user to infer the constant. When the wildcard occurs in place of a function, we synthesize a λ -abstraction, using a faster, bespoke synthesizer.

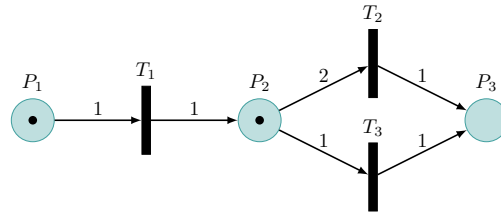
Contributions. In summary, our contributions are:

1. We develop a new unification algorithm for a subset of Haskell, that we use to replace the occurrences of the wildcard component with constants. The algorithm is generic enough for other uses, as explained in Section 8.
2. We present HOOGLE*, an extension of HOOGLE+, that synthesizes functions with constants and λ -abstractions. As shown in Section 5, it solves several problems that cannot be solved by the original HOOGLE+.

¹ Haskell list notation is represented in italic font, to avoid confusion with citations.



■ **Figure 1** A Petri net with 3 places, 3 transitions, and 6 edges. In Feng et al. [7].



■ **Figure 2** The Petri net of Figure 1 after T_1 has fired. In Feng et al. [7].

Document structure. Section 2 presents the background: Petri nets and HOOGLE+; Section 3 presents the unification algorithm; Section 4 describes the extension made to HOOGLE+; Section 5 evaluates HOOGLE*, by comparing it to HOOGLE+; Section 6 discusses the related work; Section 7 discusses the limitations of this work; and Section 8 summarizes the lessons learned and the future work.

2 Background

In this section, we describe Petri nets, HOOGLE+, and SYPET.

2.1 Petri nets

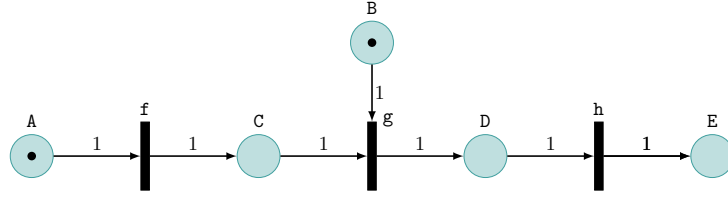
Petri nets are used by HOOGLE+ to represent the search space. In this section, we define relevant concepts of Petri nets and present examples.

► **Definition 1.** A *Petri net* is a tuple (P, T, E, W) , where P is the set of places, T is the set of transitions, $E \subseteq (P \times T) \cup (T \times P)$ is the set of edges between places and transitions and between transitions and places, and $W : E \rightarrow \mathbb{N}_0$ is a function that maps each edge to its weight. Each place in a Petri net can have zero or more tokens. A **marking** (also known as configuration) M of a Petri net $N = (P, T, E, W)$ is a function $P \rightarrow \mathbb{N}_0$ that maps each place to its number of tokens.

We represent places by drawing circles and transitions by drawing narrow rectangles. Edges are represented by arrows, and each natural number we write near each edge is its weight. For example, consider the Petri net in Figure 1. The places are P_1 , P_2 and P_3 , and the transitions are T_1 , T_2 and T_3 . The edge (P_2, T_2) has weight 2, and all the other edges have weight 1. The place P_1 has 2 tokens, whereas the remaining places have no tokens.

We explain next how transitions can change the marking of a Petri net, defining the concepts of enabled transition and firing a transition.

► **Definition 2.** We say that the transition t is **enabled** if and only if for each place p with an edge to t , the number of tokens is at least the weight of the edge (p, t) . We say that **firing an enabled transition** is changing the marking of the Petri net, consuming a certain number of tokens from each place that has an edge to the transition, and producing a certain number of tokens in each place with an edge from the transition, according to the weight of each edge.



■ **Figure 3** A Petri net for the component set $f:A \rightarrow C$, $g:B \rightarrow C \rightarrow D$, $h:D \rightarrow E$. The initial marking for the query type $A \rightarrow B \rightarrow E$.

In the example of Figure 1, T_1 is the only enabled transition. Firing T_1 produces the marking of Figure 2: one token was consumed in P_1 ($W(P_1, T_1) = 1$), and one token was produced P_2 ($W(T_1, P_2) = 1$).

The last concept about Petri nets that we introduce is a decision problem called reachability.

► **Definition 3.** Given a Petri net $N = (P, T, E, W)$, with marking M , and new marking M' , the **reachability problem for Petri nets** consists of assessing whether it is possible to reach M' starting at M and by firing a certain sequence of transitions. We say that M is the **initial marking**, M' is the **target marking**, and the **trace** is the sequence of fired transitions.

For example, consider the Petri net and marking M shown in Figure 1. A marking $\{P_1 \mapsto 0, P_2 \mapsto 0, P_3 \mapsto 1\}$ is reachable from M . A trace is $\langle T_1, T_1, T_2 \rangle$.

2.2 SyPet, Hoogle⁺ and Synthesis via Petri-net reachability

SYPET [7] is a scalable component-based synthesizer for Java and deals with large component sets by reducing the problem to a Petri-net reachability problem. HOOGLE⁺ [14] adapts this idea to the Haskell programming language, extending the approach to deal with parametric polymorphism, high-order functions, and typeclasses. In its latest version, it support input-output examples [20]. In this section, we explain how Petri nets can be used for synthesis, as well as an overview of SYPET, and the changes introduced by HOOGLE⁺.

Petri net construction. Generally speaking, SYPET starts with building a Petri net that models the component set, and then solves the reachability problem, using the resulting trace to synthesize functions. Given a component set C and a query type t , SYPET constructs the Petri net $N = (P, T, E', W)$, and the initial marking as follows.

1. The places in P are the parameter types and return types of the components in C .
2. The transitions in T are the components of C , i.e., $T = C$.
3. For each type t' , and component c , we have $(t', c) \in E$ with $W((t', c)) = m$ if and only if t' is the type of $m > 0$ parameters of c .
4. For each type t' , and component c , we have edge (c, t') if and only if t' is the return type of c .
5. For each place p that is the type of a parameter of the query type, we add a clone transition k where $W(p, k) = 1$ and $W(k, p) = 0$.
6. For each parameter type of the query type we put as many tokens as the number of arguments of the given type.

Figure 3 shows an example of a Petri net that models a synthesis problem.

Synthesizing a function from a trace. Once the Petri net is built, we solve a reachability problem, where the target marking has a single token in the place that represents the return type of the query type. Then, the resulting trace is used to synthesize the desired function. For instance, in Figure 3, the target marking would have a token in place `E`, and a trace is $\langle f, g, h \rangle$, from which the function $\backslash \text{arg1 arg2} \rightarrow h (g (f \text{ arg1}) \text{ arg2})$ is synthesized. However, synthesizing a function from a trace is not trivial, because a trace cannot distinguish between different tokens in the same place, and no notion of order of incoming edges is maintained. So, multiple functions may arise from a single trace. We do not explain how SYPET performs the reachability analysis and synthesis from traces, as it is not necessary to follow the rest of this paper; for more details, see Feng et al. [7].

Hoogle+. So far we have discussed the algorithm of SYPET, which only supports monomorphic types. However, most functions from the Haskell libraries have polymorphic types. Thus, HOOGLE+ [14] has to deal with polymorphic types, which introduce the following challenges, if we represent all the monomorphic types in the Petri net: there is no limit to the set of types that may arise (for example `[Char]`, `[[Char]]`, `[[[Char]]]`, etc.), and some components, such as `id :: a -> a`, create a transition for each place. Representing monomorphic types, even if we bound the set of types, leads to an intractable Petri net, so HOOGLE+ uses abstract types, representing sets of concrete, monomorphic types. For example, the abstract type τ is the set of all existing types, whereas `Maybe` $\tau = \{\text{Maybe } t : t \in \text{Type}\}$. The algorithm starts with the most abstract Petri net, containing only the place τ , which leads to ill-typed programs. Then, the type errors are used to refine the Petri net, introducing more concrete types. For more details, see Guo et al. [14].

3 Unification via Symbolic Execution

Petri nets allow the generation of functions with occurrences of the wildcard component. Our goal is to use HOOGLE+ to generate functions that may contain wildcards and then replace each wildcard occurrence with a constant or a custom λ -abstraction, matching the set of given input-output examples. For this purpose, we use a unification algorithm that, given a source expression with symbolic variables, and a target grounded expression, computes a substitution for the symbolic variables so that the first expression evaluates to the second expression. When a solution is found by HOOGLE+, we replace the occurrences of the wildcard with symbolic variables in the synthesized function, the parameters with the input of the input-output example, and unify the resulting expression with the output of the input-output example. Consider the example from Section 1, where constant `[0]` is appended to the input list. The query type is `[Int] -> [Int]`, and an example maps the input `[1]` to the output `[1, 0]`. The Petri net will generate $\backslash x1 \rightarrow x1 ++ \text{wildcard}$. We then replace the occurrences of `wildcard` with fresh symbols and unify `[1] ++ s1` with `[1, 0]`, where the algorithm substitutes the symbolic variable `s1` with the constant `[0]`, by inspecting the definition of `(++)`.

Requirements. The unification algorithm has the following requirements:

- The input of the unification algorithm is the function synthesized by the Petri net, as well as the input-output examples. As the algorithm inspects the definitions of the synthesized function, and of the applied components, it has to support enough language constructs to encode the component set of HOOGLE+, such as the case construct, algebraic data types, integers, or, at least naturals, ad-hoc polymorphism, function application, and λ -abstractions. Additionally, it has to support symbols both in place of constants and in place of functions.

e	$::=$	x	<i>(variable)</i>	
		$ $	s	<i>(symbolic variable)</i>
		$ $	$\lambda \bar{x}.e$	<i>(λ-abstraction)</i>
		$ $	$\mu \{\bar{e}\}$	<i>(polymorphic λ-abstraction)</i>
		$ $	$K \bar{e}$	<i>(data constructor)</i>
		$ $	case e of $\{\bar{a}\}$	<i>(case)</i>
		$ $	$e \bar{e}$	<i>(application)</i>
a	$::=$	$K \bar{x} \rightarrow e$	<i>(case alternative)</i>	

■ **Figure 4** Grammar of the supported language, λ_U .

- The unification algorithm does not need to support the occurrence of symbols, nor the application of functions or case constructs in the target expression, because this expression is always the output of an input-output example, simplifying the algorithm.

3.1 Syntax

Figure 4 presents the grammar of the language supported by the unification algorithm, which we call λ_U . Now, we discuss each construct, and present Example 4 and Example 5.

Variables play the same role as in λ -calculus, and are represented by x, x_i .

Symbolic variables denote unknown expressions and are represented by s, s_i . Symbolic variables can occur in place of functions or constants.

Abstractions have a sequence of variables (the parameters) and an expression that defines the abstraction. For example, the identity function can be encoded as $\lambda x . x$.

Polymorphic abstractions allow us to encode ad-hoc polymorphism, present in Haskell through typeclasses: each type provides an implementation for a given operation, which are chosen depending on the types of the arguments [33]. In λ_U , a polymorphic abstraction consists of a set of λ -abstractions, each one being a monomorphic variant.

Data constructors have a name and a sequence of arguments. As we do not have literals, this is the only way to represent data (natural numbers are represented using Peano numbers², such as $S (S Z)$), lists are represented using constructors *Cons* and *Nil*).

Case expressions have an expression (the *scrutinee*) and a sequence of alternatives. Each alternative has the name of a data constructor, a sequence of variables (one variable per constructor argument), and an expression. A case expression is of the form: **case** x **of** $\{Cons\ x_1\ x_2 \rightarrow Just\ x_1; Nil \rightarrow Nothing\}$. There are two differences with relation to case expressions in Haskell: we do not support guards and our alternatives only support variables after the data constructor (Haskell allows patterns such as **Just True**).

Applications have an expression and a sequence of expressions (the arguments). For example, $e\ e_1\ e_2$ denotes the application of e to the arguments e_1 and e_2 . Currying is not supported natively, and requires a specific encoding, as explained in Section 7.

► **Example 4.** Function map, applying a function f to a list l , can be encoded in λ_U as:

$$map = \lambda f\ l . \mathbf{case}\ l\ \mathbf{of}\ \{Nil \rightarrow Nil; Cons\ h\ t \rightarrow Cons\ (f\ h)\ (map\ f\ t)\}$$

² To be concise, we may write Arabic numbers as an abbreviation of the Peano representation.

► **Example 5.** We define the polymorphic abstraction eq , similar to the function `Data.Eq.(==)` from the Haskell standard library, which has two arguments, and returns *True* if and only if the arguments are equal. We define two versions: one for naturals and another for booleans: $eq = \mu \{eqN, eqB\}$.

$$\begin{aligned}
 eqN &= \lambda x_1 x_2 . \mathbf{case} x_1 \mathbf{of} \{Z \rightarrow \mathbf{case} x_2 \mathbf{of} \{Z \rightarrow True; S x_3 \rightarrow False\}; \\
 &\quad S x_3 \rightarrow \mathbf{case} x_2 \mathbf{of} \{Z \rightarrow False; S x_4 \rightarrow eq x_3 x_4\}\} \\
 eqB &= \lambda x_1 x_2 . \mathbf{case} x_1 \mathbf{of} \{False \rightarrow \mathbf{case} x_2 \mathbf{of} \{False \rightarrow True; True \rightarrow False\}; \\
 &\quad True \rightarrow \mathbf{case} x_2 \mathbf{of} \{False \rightarrow False; True \rightarrow True\}\}
 \end{aligned}$$

3.2 Inference rules

Now, we explain how the algorithm works by providing inference rules. First, we define the concept of map of substitutions, in Definition 6, and then, in Definition 7, we introduce a judgement that establishes the result of unifying two expressions. The inference rules can be used to derive judgements, and we provide different rules for different combinations of source and target expressions.

► **Definition 6.** A *map of substitutions* Σ is a mapping from symbolic variables to expressions (or applications of symbolic variables, when they occur in place of functions, as explained in Section 3.2.5). $\Sigma(s)$ denotes the value of s in map Σ , while $\Sigma[s \mapsto e]$ denotes the map Σ updated with the substitution of s with e . We write $[\]$ to denote the empty map, and $\text{dom}(\Sigma)$ to denote the set of symbolic variables that are substituted in Σ .

► **Definition 7.** The *judgement* $\Sigma \vdash e_{src} \equiv e_{tgt} \triangleright \Sigma'$, defined by the rules in Figure 5, denotes the relation where Σ' is the map of substitutions that results from unifying e_{src} and e_{tgt} , given the initial map Σ .

The rules shown in Figure 5 guarantee that for all resulting substitutions Σ' , we have $\text{eval}(\Sigma', e_{src}) = \text{eval}(\Sigma', e_{tgt})$, with $\Sigma \subseteq \Sigma'$. We validated this experimentally (a formal proof is left for future work). The evaluation function is defined in Algorithm 1. We need to use two maps Σ and Σ' because the first map represents the substitutions computed so far, and the second map represents the first one, eventually updated with new substitutions so that a substitution computed before is not discarded. So, we have $\Sigma \subseteq \Sigma'$. We will return to this topic when we address the rule for unifying data constructors, in Section 3.2.2. In the rest of this section, we present the syntax-directed rule system, which is summarized in Figure 5.

3.2.1 Unifying symbolic variables with expressions

We start with the simplest case: $\Sigma \vdash s \equiv e \triangleright \Sigma'$, in which the source expression is a symbolic variable s , and the target expression is any expression e . In this case, adding the substitution of s for e to the input map solves the problem, if s is not already assigned in the substitutions map computed so far, which is Σ (rule *SNAL*). When the symbolic variable s is already substituted in Σ , we try to unify e with $\Sigma(s)$ and add the new substitutions to the initial map (rule *SAL*).

► **Example 8.** We derive $[s_1 \mapsto s_2] \vdash s_1 \equiv False \triangleright [s_1 \mapsto s_2, s_2 \mapsto False]$.

$$\frac{s_1 \in [s_1 \mapsto s_2] \quad \frac{s_2 \notin \text{dom}([s_1 \mapsto s_2])}{[s_1 \mapsto s_2] \vdash [s_1 \mapsto s_2](s_1) \equiv False \triangleright [s_1 \mapsto s_2, s_2 \mapsto False]} \text{SNAL}}{[s_1 \mapsto s_2] \vdash s_1 \equiv False \triangleright [s_1 \mapsto s_2, s_2 \mapsto False]} \text{SAL}$$

s_1 cannot be assigned to *False* because it is already assigned to s_2 . So, we unified $\Sigma(s_1)$ with *False*.

■ **Algorithm 1** Function eval.

$$\begin{aligned}
\text{eval}(\Sigma, \lambda \bar{x} . b) &= \lambda \bar{x} . b & \text{eval}(\Sigma, \mu \{\overline{opts}\}) &= \mu \{\overline{opts}\} \\
\text{eval}(\Sigma, s) &= \begin{cases} s & \text{if } s \notin \text{dom}(\Sigma) \\ \text{eval}(\Sigma, e) & \text{if } \Sigma(s) = e \end{cases} \\
\text{eval}(\Sigma, K) &= K \\
\text{eval}(\Sigma, K e_1 \dots e_k) &= K \text{ eval}(\Sigma, e_1) \dots \text{eval}(\Sigma, e_k) \\
\text{eval}(\Sigma, f e_1 \dots e_k) &= \begin{cases} \text{eval}(\Sigma, b\{e'_1/x_1, \dots, e'_k/x_k\}) & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = \lambda x_1 \dots x_k . b \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \\ \text{eval}(\Sigma, b\{e'_1/x_1, \dots, e'_k/x_k\}) & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = \mu \{\overline{opts}\}, \\ \lambda x_1 \dots x_k . b \in \overline{opts}, \text{ and} \\ \text{eval}(\Sigma, b\{e'_1/x_1, \dots, e'_k/x_k\}) \neq \text{error} \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \\ \text{eval}(\Sigma, \Sigma(s e'_1 \dots e'_k)) & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = s \\ s e'_1 \dots e'_k \in \text{dom}(\Sigma) \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \\ s e'_1 \dots e'_k & \begin{array}{l} \text{if } \text{eval}(\Sigma, f) = s \\ s e'_1 \dots e'_k \notin \text{dom}(\Sigma) \\ \text{where } e'_i = \text{eval}(\Sigma, e_i), \text{ for } i \in \{1, \dots, k\} \end{array} \end{cases} \\
\text{eval}(\Sigma, \text{case } scr \text{ of } \{\overline{alts}\}) &= \text{eval}(\Sigma, b\{e_1/x_1, \dots, e_k/x_k\}) & \begin{array}{l} \text{if } \text{eval}(\Sigma, scr) = K e_1 \dots e_k \\ \text{and } K x_1 \dots x_k \rightarrow b \in \overline{alts} \end{array} \\
\text{eval}(\Sigma, e) &= \text{error} & \text{otherwise}
\end{aligned}$$

We have said that the target expression should not contain symbols. However, internally, we need support for symbols in the target expression, due to the rule for case constructors, which is presented in Section 3.2.3. So, there are two more rules: *SNAR* and *SAR*, similar to the *SNAL* and *SAL*, with the difference that the symbolic variable is now the target expression.

3.2.2 Unifying data constructors

The rule *DC* is applied when both expressions are data constructors, with the same data constructor and the same number of arguments, and unifies each argument of the source data constructor with the corresponding argument in the target data constructor.

► **Example 9.** We derive $[] \vdash \text{Pair } s_1 s_2 \equiv \text{Pair } \text{True } \text{Nil} \triangleright [s_1 \mapsto \text{True}, s_2 \mapsto \text{Nil}]$.

$$\frac{\frac{s_1 \notin \text{dom}([\])}{[\] \vdash s_1 \equiv \text{True} \triangleright [s_1 \mapsto \text{True}]} \quad \frac{s_2 \notin \text{dom}([s_1 \mapsto \text{True}])}{[s_1 \mapsto \text{True}] \vdash s_2 \equiv \text{Nil} \triangleright [s_1 \mapsto \text{True}, s_2 \mapsto \text{Nil}]}}{[\] \vdash \text{Pair } s_1 s_2 \equiv \text{Pair } \text{True } \text{Nil} \triangleright [s_1 \mapsto \text{True}, s_2 \mapsto \text{Nil}]}$$

where the two top rules are *SNAL*, and the bottom rule is *DC*. Both expressions are data constructors with two arguments, and the constructor is the same, *Pair*. So, we unify s_1 with *True*, and the result is passed to the unification of s_2 with *Nil*. An important aspect is

$$\begin{array}{c}
\frac{s \notin \text{dom}(\Sigma)}{\Sigma \vdash s \equiv e \triangleright \Sigma[s \mapsto e]} \text{SNAL} \qquad \frac{e'_i = \text{eval}(\Sigma, e_i), i = 1, 2, \dots, k}{\Sigma \vdash s e_1 \dots e_k \equiv \text{dst} \triangleright \Sigma[s e'_1 \dots e'_k \mapsto \text{dst}]} \text{ASNA} \\
\\
\frac{s \notin \text{dom}(\Sigma)}{\Sigma \vdash e \equiv s \triangleright \Sigma[s \mapsto e]} \text{SNAR} \qquad \frac{e'_i = \text{eval}(\Sigma, e_i), i = 1, 2, \dots, k}{\Sigma \vdash s e_1 \dots e_k \equiv \text{dst} \triangleright \Sigma'} \text{ASA} \\
\\
\frac{s \in \text{dom}(\Sigma) \quad \Sigma \vdash \Sigma(s) \equiv e \triangleright \Sigma'}{\Sigma \vdash s \equiv e \triangleright \Sigma'} \text{SAL} \qquad \frac{s \in \text{dom}(\Sigma) \quad \Sigma \vdash e \equiv \Sigma(s) \triangleright \Sigma'}{\Sigma \vdash e \equiv s \triangleright \Sigma'} \text{SAR} \\
\\
\frac{\Sigma_0 \vdash e_1 \equiv f_1 \triangleright \Sigma_1 \quad \dots \quad \Sigma_{k-1} \vdash e_k \equiv f_k \triangleright \Sigma_k}{\Sigma_0 \vdash K e_1 \dots e_k \equiv K f_1 \dots f_k \triangleright \Sigma_k} \text{DC} \qquad \frac{\lambda x_1 \dots x_k . b \in \overline{\text{opts}}}{\Sigma \vdash (\lambda x_1 \dots x_k . b) e_1 \dots e_k \equiv e' \triangleright \Sigma'} \text{AP} \\
\\
\frac{\Sigma \vdash b\{s_1/x_1, \dots, s_k/x_k\} \equiv e' \triangleright \Sigma_0 \quad s_1, \dots, s_k \text{ fresh} \quad \Sigma_0 \vdash e_1 \equiv s_1 \triangleright \Sigma_1 \quad \dots \quad \Sigma_{k-1} \vdash e_k \equiv s_k \triangleright \Sigma_k}{\Sigma \vdash (\lambda x_1 \dots x_k . b) e_1 \dots e_k \equiv e' \triangleright \Sigma_k} \text{AL} \qquad \frac{K \bar{x} \rightarrow b \in \bar{a} \quad \Sigma \vdash \text{scr} \equiv K \bar{s} \triangleright \Sigma', \bar{s} \text{ fresh} \quad \Sigma' \vdash b\{\bar{s}/\bar{x}\} \equiv e \triangleright \Sigma''}{\Sigma \vdash \mathbf{case} \text{scr} \mathbf{of} \{\bar{a}\} \equiv e \triangleright \Sigma''} \text{C}
\end{array}$$

■ **Figure 5** Syntax-directed rule system that defines the judgement $\Sigma \vdash e_{src} \equiv e_{tgt} \triangleright \Sigma'$.

that the map that results from unifying e_1 with f_1 is passed as input to unify e_2 with f_2 , and so on, to preserve all substitutions and to avoid contradictions, which is illustrated in Example 10.

► **Example 10.** We want to unify *Pair s s* with *Pair True False*. Both expressions have the same data constructor and the same number of arguments, so let us apply the *DC* rule. First, we have to derive $[\] \vdash s \equiv \text{True} \triangleright [s \mapsto \text{True}]$, by using the rule *SNAL*. Second, we have to derive $[s \mapsto \text{True}] \vdash s \equiv \text{False} \triangleright \Sigma'$, for a certain Σ' . However, this is impossible because we are trying to unify s with *False*, and the unification of the first argument substituted s with *True*.

3.2.3 Unifying case expressions with expressions

When the source expression is a case construct, the rule *C* chooses a case alternative such that the *scrutinee* (*scr*) unifies with the selected data constructor. For example, if the case expression is **case** *expr of* $\{\text{Nothing} \rightarrow \text{False}, \text{Just } x \rightarrow x\}$, and supposing that we have chosen the first alternative, we have to ensure that *expr* unifies with *Nothing*. When the

data constructor has arguments (for instance, *Just* has one argument), we generate fresh symbols (which introduces the need to support symbolic variables in the target expression). This would be the case if we selected the second alternative: we would unify *expr* with *Just s*, where *s* is a fresh symbol. Finally, we unify the body of the alternative, *b*, with the target expression, substituting the variables of the alternative with the fresh symbols ($b\{\bar{s}/\bar{x}\}$). In rule *C* of Figure 5, \bar{s} denotes the sequence of fresh symbols, \bar{x} denotes the sequence of arguments of the data constructor, and $b\{\bar{s}/\bar{x}\}$ denotes the expression *b* in which each occurrence of $x_i \in \bar{x}$ is replaced with the corresponding symbol $s_i \in \bar{s}$.

► **Example 11.** We derive

$$[] \vdash \mathbf{case\ } s \mathbf{ of\ } \{Nothing \rightarrow False, Just\ x \rightarrow x\} \equiv True \triangleright [s \mapsto Just\ s_1, s_1 \mapsto True]$$

where s_1 is a fresh symbol, and *A* and *B* are derivation subtrees. The derivation applies *C*:

$$\frac{A \quad B \quad (Just\ x \rightarrow x) \in \{Nothing \rightarrow False, Just\ x \rightarrow x\}}{[] \vdash \mathbf{case\ } s \mathbf{ of\ } \{Nothing \rightarrow False, Just\ x \rightarrow x\} \equiv True \triangleright [s \mapsto Just\ s_1, s_1 \mapsto True]}$$

where *A* abbreviates

$$\frac{s \notin \text{dom}([])}{[] \vdash s \equiv Just\ s_1 \triangleright [s \mapsto Just\ s_1], s_1 \text{ fresh}} \text{SNAL}$$

and *B* abbreviates

$$\frac{s_1 \notin \text{dom}([s \mapsto Just\ s_1])}{[s \mapsto Just\ s_1] \vdash x\{s_1/x\} \equiv True \triangleright [s \mapsto Just\ s_1, s_1 \mapsto True]} \text{SNAL}$$

We choose the alternative *Just x → x*, unify *s* with *Just* instantiated with a fresh symbolic variable s_1 and then unify the body with the target expression. Note that the symbolic variable *s* is substituted with *Just s₁* and s_1 is substituted with *True*, so we have to evaluate *s*. We have $\text{eval}([s \mapsto Just\ s_1, s_1 \mapsto True], s) = Just\ True$, and substituting *s* with *Just True* solves the unification problem.

3.2.4 Unifying λ -abstraction applications with expressions

When the source expression is an application of a λ -abstraction, and the number of arguments is the same as the number of parameters, we apply *AL*, replacing the arguments of the application with fresh symbols in *b* and unifying this result, $b\{s_1/x_1, \dots, s_k/x_k\}$, with the target expression. The idea is to propagate the target expression to the arguments of the application: the unification will compute substitutions for the fresh symbols, and then we unify each argument with the corresponding fresh symbol.

► **Example 12.** We derive $[] \vdash (\lambda x\ y . x)\ s\ F \equiv T \triangleright [s_1 \mapsto T, s \mapsto s_1, s_2 \mapsto F]$, where s_1 and s_2 are fresh symbols, and *A*, *B* and *C* are derivation subtrees.

$$\frac{A \quad B \quad C}{[] \vdash (\lambda x\ y . x)\ s\ F \equiv T \triangleright [s_1 \mapsto T, s \mapsto s_1, s_2 \mapsto F]} \text{AL}$$

where *A*, *B*, and *C* abbreviate respectively

$$\frac{s_1 \notin \text{dom}([])}{[] \vdash x\{s_1/x, s_2/y\} \equiv T \triangleright [s_1 \mapsto T], s_1, s_2 \text{ fresh}} \text{SNAL}$$

$$\frac{s \notin \text{dom}([s_1 \mapsto T])}{[s_1 \mapsto T] \vdash s \equiv s_1 \triangleright [s_1 \mapsto T, s \mapsto s_1]} \text{SNAL}$$

$$\frac{s_2 \notin \text{dom}([s_1 \mapsto T, s \mapsto s_1])}{[s_1 \mapsto T, s \mapsto s_1] \vdash F \equiv s_2 \triangleright [s_1 \mapsto T, s \mapsto s_1, s_2 \mapsto F]} \text{SNAR}$$

The λ -abstraction has two parameters, and so we generate two symbols: s_1 and s_2 . The body of the λ -abstraction is x , and $x\{s_1/x, s_2/y\} = s_1$, which is unified with the target expression. Finally, we unify the arguments s and F with s_1 and s_2 .

3.2.5 Unifying applications of symbols with expressions

As stated in Section 3.1, a symbolic variable can occur in place of a function, however, instead of assigning it directly to expressions, which is out of the scope of this work, we assign applications of symbolic variables to expressions. This allows the generation of input-output examples for unknown functions and the detection of contradictions, which is relevant for HOOGLÉ \star , as described in Section 4. For instance, unifying $\text{map } s \text{ (Cons 1 (Cons 2 Nil))}$ with $\text{Cons 2 (Cons 3 Nil)}$ generates examples for the unknown function s : assigns s 1 to 2 and s 2 to 3. On the other hand, it will be impossible to unify $\text{map } s \text{ (Cons 1 (Cons 1 Nil))}$ with $\text{(Cons 1 (Cons 2 Nil))}$, because s 1 will be assigned to 1, and then to 2, which generates a contradiction.

We have two rules *ASNA* and *ASA*, very similar to the rules for symbols, shown in Section 3.2.1. We need to store the arguments $\bar{e} = e_1 e_2 \dots e_k$ in a form as reduced as possible (using eval) because we need to compare each argument for equality³, to check if an application is already assigned as in the following example.

► **Example 13.** We derive $[s \ 0 \mapsto 2, s \ 1 \mapsto 3] \vdash s \ ((\lambda x . 0) \ 1) \equiv 2 \triangleright [s \ 0 \mapsto 2, s \ 1 \mapsto 3]$.

$$\frac{0 = \text{eval}(\Sigma, s \ ((\lambda x . 0) \ 1)) \quad \Sigma(s \ 0) = 2 \quad \Sigma \vdash 2 \equiv 2 \triangleright \Sigma}{\Sigma \vdash s \ ((\lambda x . 0) \ 1) \equiv 2 \triangleright \Sigma} \text{ASA}$$

where $\Sigma = [s \ 0 \mapsto 2, s \ 1 \mapsto 3]$. We omit the derivation of $\Sigma \vdash 2 \equiv 2 \triangleright \Sigma$. This example shows the importance of evaluating the arguments before updates and lookups to the map of substitutions. Indeed, the application $s \ ((\lambda x . 0) \ 1)$ is not substituted in Σ , but $(\lambda x . 0) \ 1$ evaluates to 0, and $s \ 0$ is already substituted in Σ .

3.2.6 Unifying applications of polymorphic abstractions with expressions

When the source expression is an application of a polymorphic abstraction, we apply *AP*, which chooses a λ -abstraction from $\overline{\text{opts}}$ and then unifies the application of this λ -abstraction to the provided arguments with the target expression. However, we cannot just choose any λ -abstraction. For instance, if the arguments are lists, we cannot choose a function that expects booleans, as the unification will fail.

► **Example 14.** We derive that $[] \vdash \text{eq } s \ \text{False} \equiv \text{True} \triangleright [s \mapsto s_1, s_1 \mapsto \text{False}, s_2 \mapsto \text{False}]$ where polymorphic abstraction *eq* was defined in Example 5.

$$\frac{\text{eq}B \in \overline{\text{opts}} \quad [] \vdash \text{eq}B \ s \ \text{False} \equiv \text{True} \triangleright [s \mapsto s_1, s_1 \mapsto \text{False}, s_2 \mapsto \text{False}]}{[] \vdash \text{eq } s \ \text{False} \equiv \text{True} \triangleright [s \mapsto s_1, s_1 \mapsto \text{False}, s_2 \mapsto \text{False}]}$$

³ Rule *ASA* can be applied only if the arguments are comparable for equality, which require the arguments not to be abstractions.

where rule AP is applied, $\overline{opts} = \{eqN, eqB\}$ and the derivation of the second hypothesis is omitted. When applying rule AP , we cannot choose the λ -abstraction eqN , because the derivation of $[\] \vdash eqN \ s \ False \equiv True \triangleright [s \mapsto s_1, s_1 \mapsto False, s_2 \mapsto False]$ gets stuck, as the case expressions of eqN do not contain alternatives matching $True$ and $False$.

3.3 Lazyness

The inference rules are lazy. For instance, consider the application $(\lambda x y . x) e_1 e_2$ being unified with $target$. We first unify the body with arguments replaced with symbols $x\{s_1/x, s_2/y\}$ with $target$, and then unify e_1 with s_1 and e_2 with s_2 . As y is not used in the abstraction, s_2 will not be assigned, so the unification of e_2 with s_2 will simply assign the symbol to the expression, and e_2 is not reduced. Thus, in principle, the algorithm supports computations with conceptually infinite structures [18]. For instance, it can unify `take sym1 (repeat sym2)` with `[3, 3]`, replacing `sym1` with 2 and `sym2` with 3.

3.4 Implementation

The unification algorithm was implemented in Haskell and performs a depth-first search: it tries to apply the rules and backtracks if it fails. The heart of symbolic execution and backtracking in the algorithm is the rule for the case construct: it attempts each alternative until it succeeds. To prevent the algorithm from running forever, we limit the depth of the DFS. The algorithm returns a substitution if found; `Mismatch`, if no substitution was found after trying all the possible rules (never reaching the limit for rules); or `DepthReached` if the maximum number of rules was reached in at least one path and no solution was found. Although the algorithm implements a search, it is very fast in practice. We conjecture that this is because the branching factor is reduced. Functions that work with lists typically have case expressions with no more than two cases (*Nil* and *Cons*), and the *Nil* case tends to be a base case (a leaf, in the search tree).

4 Extension to Hoogle \star : Hoogle \star

In this section, we describe the implementation of `HOOGLE \star` ⁴, by explaining the two independent steps presented in Section 1. We start with the modifications to introduce the wildcard component in Section 4.1, and, in Section 4.2, we address how the occurrences of the wildcard component are replaced.

4.1 The wildcard component

The first step is to add a component that matches all types so that it can occur where an integer, a list, a function, etc., is expected. `HOOGLE+` requires the name and the type of each component, so we provided the name *wildcard* associated with the type `a`, which matches all types. With this extension, `HOOGLE+` can synthesize functions containing the wildcard such as `\arg1 -> map wildcard arg1`, in which the wildcard occurs in place of a function (the first parameter of `GHC.List.map` is a function) or `\arg1 -> arg1 ++ wildcard`, in which the wildcard occurs in place of a list (both parameters of `GHC.List.++` are lists).

⁴ The `HOOGLE \star` repository is available at https://github.com/sr-lab/hoogle_plus.

The component set of HOOGLE+ contains four constants (`[]`, `True`, `False`, and `Nothing`), which are not required in HOOGLE* because the unification algorithm is able to generate them. So, these constants are not present in the default component set of HOOGLE*.

4.2 Replacing occurrences of the wildcard component

■ **Algorithm 2** Overview of HOOGLE*.

```

1: procedure HOOGLE*(components, query, N, examples)
2:   parsedExamples ← parse examples to  $\lambda_U$ 
3:   petri ← build the Petri Net with components  $\cup$  {("wildcard", "a")}
4:   for  $i = 1, \dots, N$  do
5:     synth ← synthesize a function for query using petri
6:     if synth has wildcards then
7:       completions ← COMPLETE(synth, parsedExamples, components, query)
8:       PRINT(comp) for  $comp \in completions$ 
9:     else if synth respects examples then
10:      PRINT(synth)

```

Algorithm 2 shows an overview of HOOGLE*. It takes four parameters: *components*, the component set; *query*, the query type; *N*, the number of functions the Petri net should synthesize; and *examples*, the input-output examples. HOOGLE* starts by parsing each input-output example to a pair $(x_{i1} \dots x_{ik}, y_i)$, containing a sequence of k inputs and output in λ_U , and the Petri net is built, considering the wildcard component. Then the Petri net synthesizes N functions. The function COMPLETE then tries to replace the wildcards (Algorithm 3).

4.2.1 The Complete function

COMPLETE takes four parameters: f , the function generated by the Petri net, expressed in the Hoogle+ grammar; *examples*, the examples expressed in λ_U ; *components*, the component set; and *type*, the query type. It has three main steps, presented afterward.

Step 1: Convert to λ_U . COMPLETE starts by converting the function generated by the Petri net to λ_U , where each wildcard is replaced with a fresh symbolic variable. The variable f' denotes the function in λ_U , and *symbols* denotes the array of generated symbols.

Step 2: Unify. The next step is to call the unification algorithm with the examples. For each example, the application of f' to the inputs is unified with the output, which requires as many calls to the unification algorithm as the number of examples. The result of all unifications, Σ , contains symbolic variables assigned to constants for the wildcards occurring in place of constants, and input-output examples for the wildcards occurring in place of a function⁵.

Note that Σ respects all the examples because the unification of each example uses the result of the unification of the previous example.

⁵ It is not guaranteed that each symbol is assigned in Σ , which can happen if its value does not impact the result of the unification. For instance, unifying *head* (*Cons* s_1 s_2) with *target* assigns s_1 to *target*, but does not assign s_2 . In this case, the incomplete function is rejected.

■ **Algorithm 3** Function COMPLETE.

```

1: procedure COMPLETE( $f, examples, components, type$ )
2:    $f', symbols \leftarrow$  convert  $f$  to  $\lambda_U$ , replace wildcards with fresh symbols, and return it
3:    $\Sigma \leftarrow []$  ▷ unify pairs of applications to outputs of examples
4:   for  $((x_1, \dots, x_k), y) \in examples$  do
5:      $\Sigma \leftarrow$  UNIFY( $\Sigma, f' x_1 \dots x_k, y$ )
6:     if  $\Sigma = \text{ERROR}$  then
7:       return Error
8:    $p \leftarrow$  the length of  $symbols$  ▷ i.e., the number of wildcards to replace
9:   for  $i = 1, \dots, p$  do
10:    if  $\exists \bar{e} : symbols[i] \bar{e} \in \text{dom}(\Sigma)$  then ▷ if  $symbols[i]$  denotes a function
11:       $wildcardType \leftarrow$  get the type of the wildcard  $i$  in function  $f$ 
12:       $\lambda \leftarrow$  SYNTH-LAMBDA( $wildcardType, f', examples, components, symbols[i]$ )
13:       $fill[i] \leftarrow \lambda$ 
14:    else if  $symbols[i] \in \text{dom}(\Sigma)$  then ▷ if  $symbols[i]$  denotes a constant
15:       $val \leftarrow$  eval( $\Sigma, symbols[i]$ )
16:       $fill[i] \leftarrow$  convert  $val$  to Haskell notation
17:    $res \leftarrow \{\}$ 
18:   for  $(e_1, \dots, e_p) \in fill[1] \times \dots \times fill[p]$  do ▷ all combinations of expressions
19:      $res \leftarrow res \cup f'\{e_1/symbols[1], \dots, e_p/symbols[p]\}$  ▷ replace wildcards
20:   return  $res$  in the grammar of HOOGLE+

```

Step 3: Replacing wildcards. After the unification, each symbolic variable is replaced with a constant or a λ -abstraction. Each iteration of the loop starting at line 9 replaces a symbol, assigning the replacement (or replacements, if there is more than one alternative) to the corresponding entry in the array $fill$. It starts with a lookup in Σ to determine the type of the expected term:

- If there is an application of the symbol, $symbols[i]$, in Σ , the corresponding wildcard must be replaced with a function. In this case, SYNTH-LAMBDA (Algorithm 4, Section 4.2.2), is called to synthesize λ -abstractions, with the data type of the function, $wildcardType$. Note that this function returns a set of λ -abstractions, because it may find more than one function that has the specified type and respects the examples.
- If the symbol s is itself assigned in Σ , the corresponding wildcard should be replaced with a constant. In this case, $\Sigma(s)$ is the expression that replaces the wildcard. However, this expression must be evaluated, because it may contain occurrences of other symbols, as explained in Section 3.2.3. Additionally, we replace Peano numbers with Arabic numbers and *Cons/Nil* lists with Haskell-syntax lists.

At the end of the loop that starts in line 9, $fill$ has one entry for each wildcard, each one containing a set of alternative replacements. Then, in the loop starting at line 18 we compute all the combinations of replacements for each wildcard, through a cartesian product, and, for each combination, we replace the wildcards in f' , and add the resulting expression to res .

► **Example 15.** Recall the first example of Section 1. The Petri net is able to synthesize $\backslash x1 \rightarrow x1 ++ wildcard$. We start by converting both the generated function and the input-output examples to λ_U . The function corresponds to $\lambda x_1 . (++) x1 s_1$, the input of the example becomes *Cons 1 Nil*, and the output becomes *Cons 1 (Cons 0 Nil)*. The next step is to unify $(\lambda x_1 . (++) x1 s_1)(\text{Cons } 1 \text{ Nil})$ with *Cons 1 (Cons 0 Nil)*, to compute Σ . As a result, s_1 is assigned itself in Σ to an expression, so the expected term is a constant. We have $\text{eval}(\Sigma, s_1) = \text{Cons } 0 \text{ Nil}$, that corresponds to $[0]$ in Haskell notation. Finally, the wildcard is replaced with $[0]$, which produces the function $\backslash x1 \rightarrow x1 ++ [0]$.

Note the importance of having the unification algorithm supporting symbols in place of functions, described in detail in Section 3.2.5. On the one hand, it allows replacing multiple function wildcards, because they are replaced one at a time. For instance, suppose that there are two function wildcards to replace. When replacing the first wildcard, we pick a λ -abstraction and then apply the unification algorithm to the original function, with the new λ -abstraction replaced. In this case, the second symbol remains, but the unification algorithm is able to validate if the first wildcard is replaced correctly. If the algorithm could not support symbolic functions, we would have to pick all the λ -abstractions at once and try each possible combination, which would lead to a combinatorial explosion. On the other hand, supporting symbolic functions allows for saving time, because the algorithm can detect if no λ -abstraction can fill a specific wildcard. In those cases, HOOGLE \star does not waste time calling SYNTH-LAMBDA. Section 4.2.2 discusses in detail SYNTH-LAMBDA.

4.2.2 The synthesizer for λ -abstractions

■ **Algorithm 4** Function SYNTH-LAMBDA.

```

1: procedure SYNTH-LAMBDA(type, originalFun, examples, comps, symbol)
2:   lamComps  $\leftarrow$  remove Data.ByteString and high-order components from comps
3:    $y_i \leftarrow$  the i-th parameter of originalFun for  $i = 1, \dots, n$ 
4:    $x_i \leftarrow$  the i-th parameter of the  $\lambda$ -abstraction, for  $i = 1, \dots, k$ 
5:   leafs  $\leftarrow$   $\{y_1, \dots, y_n, x_1, \dots, x_k, s\}$ , s fresh
6:   exprs  $\leftarrow$  SYNTH-EXPR(type, leafs, lamComps, 0)
7:   res  $\leftarrow$   $\{\}$ 
8:   for  $e \in$  exprs do
9:     lambda  $\leftarrow$   $\lambda x_1 \dots x_k . e$ 
10:    lambda'  $\leftarrow$  REPLACE-SYMBOLS(originalFun, lambda, examples, symbol)
11:    if lambda'  $\neq$  ERROR and the type of lambda' matches type then
12:      res  $\leftarrow$  res  $\cup$   $\{lambda'\}$ 
13:  return res ▷ a list of many  $\lambda$ -abstractions with the specified type

```

The function SYNTH-LAMBDA computes a λ -abstraction that respects the input-output examples and has the specified type. We could call HOOGLE \star recursively, but we conjecture that it would lead to performance degradations, and a simpler, faster synthesizer is enough to synthesize λ -abstractions. On the one hand, it may be needed to synthesize λ -abstractions several times during a single HOOGLE+ query, but the paper that presented HOOGLE+ [14] has shown that for many problems, HOOGLE+ may take several seconds. Note that each query may require an unbounded number of synthesis of λ -abstractions because each one of the N incomplete functions may have an arbitrary number of wildcards in place of functions. On the other hand, from our experience with the Haskell programming language, λ -abstractions are simpler than other portions of code and use fewer components. So, we decided to build a faster, bespoke synthesizer that corresponds to the function SYNTH-LAMBDA.

Search space. The search space of SYNTH-LAMBDA is composed of applications of components from HOOGLE \star . To guarantee a faster synthesis, we exclude high-order functions, as well as the module **Data.ByteString** (that seems less common in λ -abstractions, from our experience as Haskell programmers), which leaves 54 popular components. The arguments of the applications can be the parameters of the λ -abstraction, parameters of the original function, symbols (to replace using the unification algorithm), and applications of components, with at

Algorithm 5 Function SYNTH-EXPR.

```

1: procedure SYNTH-EXPR(type, leafs, components, level)
2:   exprs  $\leftarrow$  {}
3:   for  $l \in \text{leafs}$  s.t. the type of  $l$  matches type do
4:     exprs  $\leftarrow$  exprs  $\cup$  { $l$ }
5:   if level < 2 then
6:     for comp  $\in$  components s.t. the return type of comp matches type do
7:       sign  $\leftarrow$  the signature of comp, with type variables replaced s.t. the return
       type matches type
8:       prms  $\leftarrow$  extract the types of the parameters from sign
9:        $p \leftarrow$  the number of parameters of comp
10:      args[ $i$ ]  $\leftarrow$  SYNTH-EXPR(prms[ $i$ ], leafs, components, level + 1) for  $i = 1, \dots, p$ 
11:      for  $(e_1, \dots, e_p) \in \text{args}[1] \times \dots \times \text{args}[p]$  do  $\triangleright$  all combinations of expressions
12:        exprs  $\leftarrow$  exprs  $\cup$  {comp  $e_1 \dots e_p$ }
13:   return exprs  $\triangleright$  a list of many expressions with the specified type

```

Algorithm 6 Function REPLACE-SYMBOLS.

```

1: procedure REPLACE-SYMBOLS(originalFun, lambda, examples, symbol)
2:   originalFun  $\leftarrow$  originalFun{lambda/symbol}
3:    $\Sigma \leftarrow []$ 
4:   for  $((x_1, \dots, x_k), y) \in \text{examples}$  do
5:      $\Sigma \leftarrow \text{UNIFY}(\Sigma, \text{originalFun } x_1 \dots x_k, y)$ 
6:     if  $\Sigma = \text{ERROR}$  then
7:       return ERROR
8:   for each symbolic variable  $s$  in lambda do
9:     val  $\leftarrow$  eval( $\Sigma, s$ )
10:    converted  $\leftarrow$  convert val to Haskell notation
11:    lambda  $\leftarrow$  lambda{converted/ $s$ }
12:   return lambda

```

most two levels, for performance reasons (e.g., in $\backslash x \rightarrow f (g x) (h x)$, the arguments of g and h cannot be applications, only variables, and constants). For example, if the Petri net synthesizes $\backslash \text{arg1} \rightarrow \text{map } \textit{wildcard} \text{ arg1}$, for the query type $[\text{Int}] \rightarrow [\text{Int}]$, the wildcard may be replaced with $\backslash x1 \rightarrow x1 + 2$, $\backslash x1 \rightarrow x1 * (\text{length } \text{arg1})$, etc., depending on the input-output examples provided by the user.

Implementation. SYNTH-LAMBDA (Algorithm 4), takes five parameters: *type*, the signature of the function to synthesize; *originalFun*, the original function generated by the Petri net, but with wildcards replaced with fresh symbols; *examples*, the input-output examples of the original function; *comps*, the component set; and *symbol*, the symbolic variable that the new λ -abstraction should replace in *originalFun*. It follows a generate-and-test approach: SYNTH-EXPR does a type-guided enumeration of λ -abstractions, and then REPLACE-SYMBOLS tests the original function with each new λ -abstraction in place of the corresponding symbol, and replace symbols if any. Finally, we check that the λ -abstraction has the specified type, because SYNTH-EXPR does not perform a *full* type-checking, and only uses types to prune the search, thus can return ill-typed expressions. At this stage, type classes are ignored, and we leave for future work the analysis of their impact on the algorithm and its performance.

SYNTH-EXPR (Algorithm 5) takes four parameters: *type*, the type of the expression to synthesize; *leafs*, the set that contains the parameters of the new λ -abstraction (x_1, \dots, x_n), the parameters of the original function (y_1, \dots, y_k), and a fresh symbol; *components*, the component set of HOOGLE \star excluding high-order functions and the `Data.ByteString` module; and the depth of applications, *level*. If *level* is equal or greater than 2, SYNTH-EXPR only returns the *leafs* whose type matches *type*, to ensure that the maximum level is not exceeded. Otherwise, it returns also the application of components whose return type matches *type*, and the arguments are synthesized by calling SYNTH-EXPR recursively. Note that we may have to replace type variables, which we do in line 7. For instance, if the component has type `a -> a` and *type* is `Int`, we replace `a` with `Int`.

REPLACE-SYMBOLS (Algorithm 6) takes four parameters: the original Hoogle+ function, *originalFun*; the new λ -abstraction, *lambda*; the input-output examples, *examples*; and the symbolic variable that *lambda* replaces. It starts by replacing *symbol* with *lambda* in the original function. Then, the unification algorithm is used as in the COMPLETE function: for each example, we unify the application of *originalFun* to the inputs of the example with the expected output. Finally, each symbol that belongs to λ is replaced with a *lookup* in Σ , as COMPLETE does. Note that every symbol in the new λ -abstraction must be a constant because the component set excludes high-order functions.

Example 16 illustrates the synthesis of wildcards in place of functions.

► **Example 16.** Recall the second example of Section 1. The Petri net is able to synthesize `\x1 -> map wildcard x1`, which corresponds to $\lambda x_1 . map\ s\ x_1$ in λ_U . The example is converted to λ_U , and the unification is performed, assigning applications of *s* (for instance, *s* 1 to 1), which informs that the expected term is a λ -abstraction. Then, we call SYNTH-LAMBDA, where *type* is `Int -> Int`, *originalFun* is $\lambda x_1 . map\ s\ x_1$, *examples* is $\{([1, 2, 3]), [1, 4, 9])\}$ and *symbol* is *s*. The *leafs* are the parameter of the original function (x_1), the parameter of the new λ -abstraction (y_1), and a fresh symbol. One of the generated λ -abstractions can be `\y1 -> (GHC.Num.*) y1 y1`. Then REPLACE-SYMBOLS unifies `(\x1 -> map (\y1 -> y1 * y1) x1) [1, 2, 3]` with `[1, 4, 9]`, which succeeds. Finally, we check that the function has type `Int -> Int`.

5 Evaluation

In this chapter we empirically evaluate HOOGLE \star , answering the following research questions:

RQ1 Can HOOGLE \star solve all the problems that HOOGLE+ solves, without performance degradation?

RQ2 Can HOOGLE \star solve more problems than HOOGLE+?

5.1 Evaluation Design

Benchmarks. We use two different sets of benchmarks. To answer RQ1, we use the first set of 44 benchmarks (Table 1) from the original paper of HOOGLE+ [14], which consists of only query types. To answer RQ2, we use the second set of 26 benchmarks (Table 2), which consists of a query types and input-output examples and requires the generation of constants or λ -abstractions⁶.

⁶ Most of those benchmarks were adapted from questions in StackOverflow because we could not use them directly (e.g., if a question used floats, we changed, when possible, to integers). We systematically searched StackOverflow for Haskell problems, excluding the ones that did not require the generation of

■ **Table 1** First set of 44 benchmarks (from HOOGLE+ [14]).

#	Problem	Query
1	firstRight	[Either a b] -> Either a b
2	firstKey	[(a, b)] -> a
3	flatten	[[[a]]] -> [a]
4	repl-funcs	(a -> b) -> Int -> [a -> b]
5	containsEdge	Int -> (Int, Int) -> Bool
6	multiApp	(a -> b -> c) -> (a -> b) -> a -> c
7	appendN	Int -> [a] -> [a]
8	pipe	[a -> a] -> (a -> a)
9	intToBS	Int64 -> ByteString
10	cartProduct	[a] -> [b] -> [(a, b)]
11	applyNtimes	(a -> a) -> a -> Int -> a
12	firstMatch	[a] -> (a -> Bool) -> a
13	mbElem	Eq a => a -> [a] -> Maybe a
14	mapEither	(a -> Either b c) -> [a] -> ([b], [c])
15	hoogle01	(a -> b) -> [a] -> b
16	zipWithResult	(a -> b) -> [a] -> [(a, b)]
17	splitStr	String -> Char -> String
18	lookup	[(a, b)] -> a -> b
19	fromFirstMaybes	a -> [Maybe a] -> a
20	map	(a -> b) -> [a] -> [b]
21	maybe	Maybe a -> a -> Maybe a
22	rights	[Either a b] -> Either a [b]
23	mbAppFirst	b -> (a -> b) -> [a] -> b
24	mergeEither	Either a (Either a b) -> Either a b
25	test	Bool -> a -> Maybe a
26	multiAppPair	(a -> b, a -> c) -> a -> (b, c)
27	splitAtFirst	a -> [a] -> ([a], [a])
28	2partApp	(a->b)->(b->c)->[a]->[c]
29	areEq	Eq a => a -> a -> Maybe a
30	eitherTriple	Either a b -> Either a b -> Either a b
31	mapMaybes	(a -> Maybe b) -> [a] -> Maybe b
32	head-rest	[a] -> (a, [a])
33	appBoth	(a -> b) -> (a -> c) -> a -> (b, c)
34	applyPair	(a -> b, a) -> b
35	resolveEither	Either a b -> (a->b) -> b
36	head-tail	[a] -> (a,a)
37	indexesOf	([(a,Int)] -> [(a,Int)]) -> [a] -> [Int] -> [Int]
38	app3	(a -> b -> c -> d) -> a -> c -> b -> d
39	both	(a -> b) -> (a, a) -> (b, b)
40	takeNdropM	Int -> Int -> [a] -> ([a], [a])
41	firstMaybe	[Maybe a] -> a
42	mbToEither	Maybe a -> b -> Either a b
43	pred-match	[a] -> (a -> Bool) -> Int
44	singleList	Int -> [Int]

Experiments. We compare HOOGLE* to the original HOOGLE+ as described below, giving each benchmark a timeout of 60 seconds, in the first set of benchmarks, and 90 seconds in the second set.

1. We run both HOOGLE+ (twice, with and without the constants `True`, `False`, `Nothing` and `[]`) and HOOGLE* on the 44 original benchmarks measuring the number of synthesized solutions, and the time taken to synthesize the first solution. For each benchmark, we

constants and λ -abstractions, and problems exercising the same capabilities. We also excluded problems that could not be solved by Hoogle+, for other reasons than the absence of constants and λ -abstractions. To diversify the components used we searched for questions using specific components. No problem that we have excluded would be solved by Hoogle+.

■ **Table 2** Second set of 26 benchmarks.

#	Problem	Query	Examples
1	mapAdd	[Int] -> [Int]	[[[1, 2, 3], [2, 3, 4]]]
2	mapSquare	[Int] -> [Int]	[[[1, 2]], [1, 4]]
3	appendConst	[Int] -> [Int]	[[[1, 2, 3], [1, 2, 3, 1000]]]
4	filterDiff	[Int] -> [Int]	[[[1, 2, 3], [1, 3]]]
5	getFirstOnes	[Int] -> [Int]	[[[1, 1, 0, 1, 2]], [1, 1]]
6	removeFirstOnes	[Int] -> [Int]	[[[1, 1, 0, 0, 1, 2]], [0, 0, 1, 2]]
7	listIntersect	[Int] -> [Int] -> [Int]	[[[0, 2, 4], [2, 4, 6]], [2, 4]]
8	indexConst	[a] -> a	[[[1, 2, 0, 3, 0, 1]], 3] [[2, 3, 4], True), [[2, 1, 4], False]]
9	allGreaterThan	[Int] -> Bool	[[[0, 0, 4, 4, 3]], [4, 3]]
10	dropConst	[Int] -> [Int]	[[[2, 0, 1, 3]], [2, 3]]
11	filterGreaterThan	[Int] -> [Int]	[[[1, 2], (2, 2), (3, 0)], [(2, 2)]]
12	filterPairs	[(Int, Int)] -> [(Int, Int)]	[[[1, 2, 1, 3, 4, 4]], [1, 1]]
13	filterEq	[Int] -> [Int]	[[1], [1, 1]]
14	replicateConst	Int -> [Int]	[[[1, 2, 3], [3, 4, 5]], [4, 6, 8]]
15	addElemsTwoLists	[Int] -> [Int] -> [Int]	[[[1, 3, 1]], 11]
16	sumSquares	[Int] -> Int	[[[1, 3, 2]], [1, 2]]
17	removeMax	[Int] -> [Int]	[[True, True], False), [[False, False], True), [[True, False], True), [[False, True], True)] [[[False, False], True), [[True, False], False), [[True], True)]
18	nandPair	(Bool, Bool) -> Bool	[[[1, 3]], [[3, 1]]] [[[Nothing, Just 1]], False), [[Just 0, Just 0]], True), [[[Just 0, Nothing]], False]]
19	allEqBool	[Bool] -> Bool	[[[(True, True), (False, False)], False), [(True, True), (False, False), (True, True)], False), [[True, True), (True, True)], True), [(False, False)], False]]
20	mapReverse	[a] -> [[a]]	[[[1, 2], 3]]
21	allJust	[Maybe a] -> Bool	[[[1, 2, 3], [1.5, 2.5, 3.5]]] [[[100, 200, 300]], [120, 220, 320]]]
22	andListPairs	[(Bool, Bool)] -> Bool	
23	sumPairEntries	(Int, Int) -> Int	
24	filterPairsTyClass	(Eq a) => [(a, a)] -> [(a, a)]	
25	mapAddFloat	[Float] -> [Float]	
26	mapAddLarge	[Int] -> [Int]	

ask both synthesizers to synthesize at most 10 solutions (parameter N in Algorithm 2). The goal is to understand the impact of the addition of the wildcard component, and the removal of the constant components.

- We run both HOOGLÉ \star and the version of HOOGLÉ $+$ that supports examples[20], on the 26 benchmarks, measuring also the time consumed replacing the occurrences of the wildcard, and we ask both synthesizers to synthesize at most 35 solutions⁷.

Experimental setup. We run the experiments on a laptop with an AMD 5600G, running at 3.9 GHz, with 6 cores and 16 GB of RAM. All the versions of HOOGLÉ $+$ and HOOGLÉ \star use only two cores. The operating system is Ubuntu 22.04.2 LTS, the version of stack is 2.9.1, and

⁷ This is a higher value than in the previous step, because many of the N functions synthesized by the Petri net may be rejected due to the test of input-output examples, and with a lower value of N , both synthesizers ended the search before the timeout, without finding any solution.

the version of GHC is 8.4.4. The component set used by both HOOGLE+ versions contains the following modules: `Data.Bool`, `Data.ByteString.Builder`, `Data.ByteString.Lazy`, `Data.Either`, `Data.Eq`, `Data.Function`, `Data.Int`, `Data.Maybe`, `Data.Ord`, `Data.Tuple`, `GHC.Char`, `GHC.List`, and `Text.Show`. The total number of components is 297. The component set used by HOOGLE \star is the same, except that we removed the constants `Data.Bool.True`, `Data.Bool.False`, `Data.Maybe.Nothing` and `[]`, and we added the wildcard component. Those constants can be synthesized by the unification algorithm, so the component set does not need to contain them.

5.2 Results

In this section, we discuss the results of the experiments performed to answer the research questions stated at the beginning of the Section 5.

Results of the first set of benchmarks. The results of the first set of benchmarks are presented in Table 3, which shows that HOOGLE \star solves two problems that HOOGLE+ could not solve (benchmarks 6 and 9), and HOOGLE+ one problem that HOOGLE \star could not solve (benchmark 35). We have not found significant differences in the synthesized solutions, and in most benchmarks, there are solutions in common.

HOOGLE \star tends to be faster at synthesizing the first solution and synthesizes more solutions. On average, HOOGLE \star synthesizes 2.95 solutions per benchmark and takes 3.92 seconds to synthesize the first solution. HOOGLE+ synthesizes 2.41 solutions and takes 5.58 seconds. This can be explained by the removal of the four constants: on average, HOOGLE+ without constants takes 3.37 seconds to synthesize the first solution, so, in average, it is faster than HOOGLE+ with constants and HOOGLE \star . Indeed, the removal of the constants leads to a smaller component set, however, the reason for that is not the number of components that were removed, but the kind of components. Note that in the Petri net encoding, constants correspond to nullary transitions, i.e., transitions that do not need tokens to fire, so they can fire at any moment, leading to a higher branching factor. Thus, the removal of a constant should have more impact than the removal of a function.

Results of the second set of benchmarks. The results of the second set of benchmarks are shown in Table 4. HOOGLE \star solves 22 out of 26 benchmarks, whereas HOOGLE+ solves only 3 (benchmarks 50, 52, and 62), which are all solved by HOOGLE \star . This happens because most benchmarks require constants and λ -abstractions to be synthesized, which HOOGLE+ is not able to do. The authors of HOOGLE+ [14] argue that the absence of λ -abstractions does not impact the completeness of the method, because terms with λ -abstractions can be replaced with a term in point-free style, using the combinators S, K and I. However, this requires adding a nullary version of each component to the component set, which the authors consider infeasible, and in practice, only a small subset is added. The component sets of each version of HOOGLE+ used in our evaluation contain the combinators S, K, and I (module `Data.Function`), but it was not enough to solve the problems that require λ -abstractions.

In the benchmarks that require the synthesis of constants, the time spent completing the functions is always lower than 20% of the total time. However, in the benchmarks that require the synthesis of λ -abstractions, the time spent completing the wildcards can reach more than 50% of the total time, as happens in benchmarks 53, 59, and 60. HOOGLE \star cannot solve benchmark 51, whose solution is `\arg1 arg2 -> filter (\x1 -> x1 'elem' arg2) arg1`, because it requires the Petri net to synthesize the incomplete function `\arg1 arg2 -> filter wildcard arg1`, which does not use `arg2`, and the Petri net always synthesizes functions that use all the parameters. It also fails to solve benchmark 68, which is very similar to 56, with the difference that

■ **Table 3** Results of the first set of benchmarks. For both synthesizers, we show the number of solutions and the total time to synthesize the first solution, in seconds, or - if no solution was produced within the timeout of 60 seconds.

#	Benchmark	Hoogle+		Hoogle+, no consts.		Hoogle*	
		Time (s)	Sols.	Time (s)	Sols.	Time (s)	Sols.
1	firstRight	0.56	5	0.53	5	0.47	6
2	firstKey	2.32	4	1.41	2	1.21	2
3	flatten	1.09	9	6.10	9	0.93	9
4	repl-funcs	0.81	2	0.57	2	0.5	5
5	containsEdge	0.92	2	0.82	1	0.66	1
6	multiApp	-	0	-	0	1.73	2
7	appendN	0.6	10	0.54	10	0.48	10
8	pipe	6.99	4	7.48	2	7.41	2
9	intToBS	-	0	-	0	0.66	6
10	cartProduct	20.08	1	3.97	1	1.43	1
11	applyNtimes	4.88	2	5.06	3	5.15	6
12	firstMatch	0.97	5	1.03	5	1.23	6
13	mbElem	-	0	-	0	-	0
14	mapEither	2.28	1	7.42	1	3.07	1
15	hoogle01	0.68	4	0.66	4	0.61	9
16	zipWithResult	-	0	-	0	-	0
17	splitStr	0.58	5	0.54	4	0.5	9
18	lookup	-	0	-	0	-	0
19	fromFirstMaybes	2.17	3	2.03	5	1.37	2
20	map	0.78	5	0.81	5	0.54	7
21	maybe	0.68	1	0.69	1	0.51	1
22	rights	30.41	1	16.18	1	6.64	1
23	mbAppFirst	1.31	1	0.98	1	0.85	1
24	mergeEither	-	0	-	0	-	0
25	test	10.68	2	9.23	1	12.98	1
26	multiAppPair	-	0	-	0	-	0
27	splitAtFirst	1.01	5	0.79	1	0.72	3
28	2partApp	2.08	5	4.08	3	2.64	3
29	areEq	-	0	-	0	-	0
30	eitherTriple	-	0	-	0	-	0
31	mapMaybes	0.72	5	0.70	6	0.57	9
32	head-rest	3.79	3	8.67	3	2.36	3
33	appBoth	1.82	1	4.56	1	1.6	1
34	applyPair	1.68	1	1.98	1	3.82	1
35	resolveEither	42.49	1	-	0	-	0
36	head-tail	9.69	2	10.37	3	11.03	2
37	indexesOf	22.38	1	-	0	54.35	1
38	app3	0.59	1	0.86	1	0.52	7
39	both	-	0	-	0	-	0
40	takeNdropM	-	0	-	0	-	0
41	firstMaybe	1.71	6	1.41	8	1.31	4
42	mbToEither	-	0	-	0	-	0
43	pred-match	1.02	4	0.97	4	0.9	4
44	singleList	0.66	4	0.61	3	0.51	4
average		5.58	2.41	3.37	2.20	3.92	2.95

the query type has a typeclass constraint, instead of a monomorphic type. The solution is `\arg1 -> filter (\p -> fst p == snd p) arg1`, however, the Petri net does not synthesize the incomplete function `\arg1 -> filter wildcard arg1` within the timeout (whereas it synthesizes when the type is monomorphic). Benchmark 69 uses real numbers, that are not supported by the unification algorithm, and benchmark 70 contains input-output examples with large constants, leading the unification to reach the maximum depth before finding valid assignments. Comparing the solutions synthesized for the benchmarks that HOOGLE+ solves, the solutions of HOOGLE* are simpler, using fewer components. For instance, in benchmark 52, HOOGLE+ synthesizes `\arg0 -> last (init (init arg0))`, whereas HOOGLE* synthesizes `\arg1 -> (!!)` `arg1 3`.

■ **Table 4** Results of the second set of benchmarks. This table shows, for both synthesizers, the time elapsed to synthesize the first solution, in seconds, as well as the number of solutions, and the time spent replacing symbols until the first solution is completed. The timeout is 90 seconds.

#	Benchmark	Hoogle+ with examples		Hoogle \star		
		Time (s)	Sols.	Time (s)	Unify (s)	Sols.
45	mapAdd	-	0	8.07	0.66	11
46	mapSquare	-	0	7.99	0.61	11
47	appendConst	-	0	4.14	0.49	1
48	filterDiff	-	0	14.26	6.04	10
49	getFirstOnes	-	0	1.89	0.16	21
50	removeFirstOnes	2.55	1	1.49	0.18	22
51	listIntersect	-	0	-	-	0
52	indexConst	3.94	1	1.16	0.18	1
53	allGreaterThan	-	0	21.45	15.42	25
54	dropConst	-	0	1.81	0.18	9
55	filterGreaterThan	-	0	15.14	6.48	10
56	filterPairs	-	0	2.26	0.19	6
57	filterEq	-	0	24.77	11.5	12
58	replications	-	0	1.18	0.19	14
59	addElemsTwoLists	-	0	74.56	66.52	10
60	sumSquares	-	0	25.58	20.22	10
61	removeMax	-	0	14.27	5.96	10
62	nandPair	30.95	4	8.91	3.84	10
63	allEqBool	-	0	7.33	1.63	20
64	mapReverse	-	0	6.0	0.73	10
65	allJust	-	0	17.32	1.14	8
66	andListPairs	-	0	7.9	1.05	20
67	sumPairEntries	-	0	8.01	0.67	27
68	filterPairsTyClass	-	0	-	-	0
69	mapAddFloat	-	0	-	-	0
70	mapAddLarge	-	0	-	-	0
average		12.48	0.23	12.52	6.55	10.70

5.3 Answers to Research Questions

Given the results discussed in Section 5.2, we answer the two research questions as follows:

RQ1 The addition of the wildcard component did not lead to performance degradations.

Instead, the removal of constants resulted in performance improvements. From the original HOOGLE+ benchmarks, there is a single benchmark that HOOGLE+ solves and HOOGLE \star cannot solve within the timeout, but it solves two that HOOGLE+ does not solve.

RQ2 HOOGLE \star can solve many more new problems than HOOGLE+, especially when constants or λ -abstractions are required, which makes it able to solve new classes of problems.

We also found that in the cases that both synthesizers produce solutions, the solutions of HOOGLE \star are simpler, since they use fewer components.

6 Related Work

In this section, we compare our work to other research in program synthesis, unification, and symbolic execution. Most of the related work has been already presented in the HOOGLE+ original paper, so our focus is the work apart from this one.

6.1 Program Synthesis

Hoogle+ related work summary. The subjects most directly related to HOOGLE+ are type inhabitation and graph reachability. However, most of the related work on type inhabitation is based on classical proof search, such as AGDA [30], or produce solutions

that do not use all the arguments, such as DJINN [1]. In turn, the related work on graph reachability only supports functions with a single parameter, such as PROSPECTOR [25], or does not support polymorphism, such as SYPET [7]. When compared to other API search tools, such as HOOGLER [26], HOOGLER+ is able to synthesize applications of multiple components. Using statistical methods to improve the search, such as SLANG [34], the authors of HOOGLER+ conjecture that it is not effective in functional languages, due to the “high degree of compositionality”. There are also approaches to scalable proof search; however, the search space is restricted to names of parameters, functions, or fields [32], or does not support polymorphism, such as INSYNTH [15].

Synthesis from sketches. The idea of completing programs with holes, also known as sketches, has already been used in SKETCH [36] and ROSETTE [37], in which SAT/SMT solvers infer integer constants. However, in our work, a hole can be replaced with an expression of any algebraic type, or λ -abstractions. More recently, SMYTH [24], an evaluator-based program synthesizer, replaces holes with any expression, including case expressions, by performing a search guided by input-output examples. However, it inherits scalability issues from MYTH [31], the base of SMYTH, and the authors consider that HOOGLER+ “might also be incorporated into our approach in future work”. SCRYBE [27] extends the approach of SMYTH, with example propagation, and can solve more problems than SMYTH. However, we conjecture that the scalability issues remain, as the evaluation uses specific component sets for each test of at most 10 components [28], whereas the component set of HOOGLER+ has 291 components. GHC, a Haskell compiler, supports programs with missing expressions, suggesting valid fits [9]. However, constants are excluded (apart from already defined constants, such as `True`) and λ -abstractions. PROPR [10] uses this GHC feature to replace faulty sub-expressions on Haskell programs, and suggest constants, that, however, are limited to the ones contained in the program to repair.

Component-based synthesis. Apart from the related work of HOOGLER+, PETSy [38] performs a top-down enumerative search, instead of using a Petri net encoding. Its evaluation shows that, at least with 130 components, its performance is comparable to HOOGLER+. However, it does not synthesize constants. HECTARE [23], a new synthesizer for Haskell that uses a new graph data structure to represent the search space, has shown to be faster than HOOGLER+, but it does not support constants nor λ -abstractions.

6.2 Unification and Symbolic Execution

***E*-Unification.** Unification is a process that, given two expressions, tries to replace the symbols in both expressions, such that the resulting expressions are syntactically equal [2]. In our case, the goal is to make two expressions equal after evaluation. This leads us to *E*-Unification, in which the equality of terms is established by a set of equations *E*: two terms *s*, *t* are equal if and only if $s \approx t \in E$ [35]. There are several approaches to solving *E*-Unification [8, 6], but we have not found any formulation that could be directly applied to our context. The same can be stated about Huet’s algorithm [19], which solves the unification problem for typed λ -calculus, from which Haskell’s Core language is an extension [21].

Symbolic execution. Symbolic execution tools explore multiple paths of a program to find counterexamples for a given property [3] and the unification problem discussed in this article can be reduced to finding a counterexample for $e_{src} \neq e_{tgt}$. G2 [16] and G2Q [17] are two symbolic execution tools for Haskell, but they do not support symbolic variables in place of

functions, which is required for HOOGLE \star . NEBULA [22], built on top of G2, supports symbols in place of functions, and treats applications of symbolic variables in a way similar to our approach: it replaces the application with a fresh symbol denoting the return value. However, it does not fully evaluate the arguments, so it may treat two equivalent calls as different calls. NEBULA can prove the equivalence of Haskell programs, by combining symbolic execution and coinduction, whereas our algorithm only finds assignments to symbolic variables. However, it is one order of magnitude faster, which makes the difference in the performance of HOOGLE \star . SCV [29] uses symbolic execution to validate software contracts in Racket programs and supports symbols in place of functions, but instead of assigning applications of functions to expressions, it generates candidate functions. However, while our algorithm supports infinite structures, SCV does not, since Racket is a strict language.

7 Limitations

Using polymorphic abstractions, instead of the standard way of implementing typeclasses, dictionary passing [33], simplifies the algorithm, especially when the term that determines the version of the polymorphic function is a symbol. But, as a drawback, this approach requires that each time a new monomorphic variant of a polymorphic operation is provided, the existing code must be edited (the new implementation must be added to each occurrence of the corresponding polymorphic abstraction). However, since the component set is not expected to change, this does not impact the usage of HOOGLE \star .

Currying is not supported for practical reasons. Whenever a curried application is translated to λ_U , we need to replace it with a λ -abstraction: supposing that f takes n arguments, we rewrite $f e_1 \dots e_m$ as $\lambda x_{m+1} \dots x_n . f e_1 \dots e_m x_{m+1} \dots x_n$ (with $m < n$). Also, for practical reasons, data constructors are not treated as the left side of abstractions, which means that a data constructor cannot be used as a function directly.

Data is represented by data constructors, which simplifies the algorithm, because all operations can be written in λ_U and each value can be built incrementally, by choosing a branch of each case expression. For instance, if we had to use the constant representation of integers, the implementation of operators such as integer comparison could not be expressed in λ_U , and expressions such as $n1 \leq n2$ would have to be processed by an SMT solver. A drawback of this representation is that real numbers are not supported (benchmark 69), and, in some specific cases, large integers may lead to an intractable search (if it is required to iterate the whole structure). Unifying a symbol with a large number, which is the case of benchmark 47, simply requires the application of *SNAL* or *SAL*; however, unifying $s + 1$ with N (similar to what happens in benchmark 70) requires a depth greater than N , which, in the context of complex problems with large branching factors, may become intractable.

Allow unused parameters. The Petri net does not synthesize functions that do not use all parameters, but the wildcards could be replaced with expressions using the remaining parameters. For instance, HOOGLE \star cannot synthesize `\xs n -> filter (\x -> x < n) xs`, because `\xs n -> filter wildcard xs` does not use the parameter `n`.

Queries with typeclass constraints are not solved, as in benchmark 68, because the Petri net becomes significantly slower when there are typeclass constraints (typeclass constraints are treated as extra arguments of the type query).

Completeness, normal form, and soundness. We do not have a definition of normal form for the terms of λ_U , nor proofs of completeness and of the guarantees of the inference rules, stated in Section 3.2.

8 Conclusion

In this work we developed a unification algorithm for a subset of the Haskell programming language and extended HOOGLE+, which can now synthesize constants and λ -abstractions.

Unification algorithm. To evaluate HOOGLE*, we have encoded 92 functions from the Haskell standard library⁸ in λ_U , and our algorithm successfully replaced the occurrences of the wildcard component for constants. But it has other applications; for instance, it can be used to compute inverses (by unifying $f\ s_1 \dots s_k$ with the output, $s_1 \dots s_k$ will be assigned to the values of the arguments), or for software testing and verification, finding counterexamples (for instance, if a function f is expected to always return a positive number, we can unify the application of f to symbolic variables with 0, to search for inputs that eventually make the function return 0).

Hoogle*. HOOGLE* can solve more problems than the original HOOGLE+ as it successfully synthesizes constants and λ -abstractions, without performance degradation. As explained in Section 6, existing synthesizers do not synthesize constants and λ -abstractions, or do not have the scalability that Petri nets give to HOOGLE+. HOOGLE* can generate constants and λ -abstractions while maintaining the scalability of Petri nets. Although we extended HOOGLE+, the contributions are not exclusive to this synthesizer, as they can be applied to other Petri-net synthesizers, such as SYPET. As a program synthesizer, it can impact science and industry in different ways: discovering new algorithms, allowing end users to build programs, improving teaching or assisting programmers [11, 5].

Future work. The main lines of future work are: supporting the representation of real numbers, as well as large integers; allowing the Petri net to synthesize functions that do not use all parameters; improving the synthesis of queries involving typeclass constraints; providing notions of completeness, normal forms, and a proof of the guarantees claimed in Section 3.2; and incorporating typeclasses in the type-checker of SYNTH-EXPR.

References

- 1 Lenart Augustsson. Djinn. URL: <https://github.com/augustss/djinn>.
- 2 Franz Baader. Unification theory. In Klaus U. Schulz, editor, *Word Equations and Related Topics, First International Workshop, IWWERT '90, Tübingen, Germany, October 1-3, 1990, Proceedings*, volume 572 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 1990. doi:10.1007/3-540-55124-7_5.
- 3 Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018. doi:10.1145/3182657.
- 4 João Costa Seco, Jonathan Aldrich, Luís Carvalho, Bernardo Toninho, and Carla Ferreira. Derivations with holes for concept-based program synthesis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2022, pages 63–79, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3563835.3567658.


⁸ From the modules `Data.Maybe`, `Data.Either`, `Data.Bool`, `GHC.List`, `Data.Ord` and `GHC.Num`.

- 5 Cristina David and Daniel Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20150403, 2017. doi:10.1098/rsta.2015.0403.
- 6 Daniel J. Dougherty and Patricia Johann. An improved general e-unification method. *J. Symb. Comput.*, 14(4):303–320, 1992. doi:10.1016/0747-7171(92)90010-2.
- 7 Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017. doi:10.1145/3009837.3009851.
- 8 Jean H. Gallier and Wayne Snyder. A general complete E -unification procedure. In Pierre Lescanne, editor, *Rewriting Techniques and Applications, 2nd International Conference, RTA-87, Bordeaux, France, May 25-27, 1987, Proceedings*, volume 256 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 1987. doi:10.1007/3-540-17220-3_19.
- 9 Matthías Páll Gissurarson. Suggesting valid hole fits for typed-holes (experience report). In Nicolas Wu, editor, *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 179–185. ACM, 2018. doi:10.1145/3242744.3242760.
- 10 Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. PROPR: property-based automatic program repair. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1768–1780. ACM, 2022. doi:10.1145/3510003.3510620.
- 11 Sumit Gulwani. Dimensions in program synthesis. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 13–24. ACM, 2010. doi:10.1145/1836089.1836091.
- 12 Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Found. Trends Program. Lang.*, 4(1-2):1–119, 2017. doi:10.1561/2500000010.
- 13 Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 122–136. ACM, 2022. doi:10.1145/3519939.3523450.
- 14 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL):12:1–12:28, 2020. doi:10.1145/3371080.
- 15 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38. ACM, 2013. doi:10.1145/2491956.2462192.
- 16 William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 411–424. ACM, 2019. doi:10.1145/3314221.3314618.
- 17 William T. Hallahan, Anton Xue, and Ruzica Piskac. G2Q: haskell constraint solving. In Richard A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, pages 44–57. ACM, 2019. doi:10.1145/3331545.3342590.
- 18 Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *ACM SIGPLAN Notices*, 27(5):1, 1992. doi:10.1145/130697.130698.

- 19 Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975. doi:10.1016/0304-3975(75)90011-0.
- 20 Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):205:1–205:27, 2020. doi:10.1145/3428273.
- 21 SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.
- 22 John C. Kolesar, Ruzica Piskac, and William T. Hallahan. Checking equivalence in a non-strict language. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1469–1496, 2022. doi:10.1145/3563340.
- 23 James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. Searching entangled program spaces. *Proc. ACM Program. Lang.*, 6(ICFP):23–51, 2022. doi:10.1145/3547622.
- 24 Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.*, 4(ICFP):109:1–109:29, 2020. doi:10.1145/3408991.
- 25 David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 48–61. ACM, 2005. doi:10.1145/1065010.1065018.
- 26 Neil Mitchel. Hoogle. URL: <https://hoogle.haskell.org/>.
- 27 Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. Program synthesis using example propagation. *CoRR*, abs/2210.13873, 2022. doi:10.48550/arXiv.2210.13873.
- 28 Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. Scrybe. <https://github.com/NiekM/scrybe>, 2022.
- 29 Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *J. Funct. Program.*, 27:e3, 2017. doi:10.1017/S0956796816000216.
- 30 Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijten, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008. doi:10.1007/978-3-642-04652-0_5.
- 31 Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630. ACM, 2015. doi:10.1145/2737924.2738007.
- 32 Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286. ACM, 2012. doi:10.1145/2254064.2254098.
- 33 John Peterson and Mark P. Jones. Implementing type classes. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 227–236. ACM, 1993. doi:10.1145/155090.155112.
- 34 Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 419–428. ACM, 2014. doi:10.1145/2594291.2594321.
- 35 Jörg H. Siekmann. Unification theory. *J. Symb. Comput.*, 7(3/4):207–274, 1989. doi:10.1016/S0747-7171(89)80012-4.

- 36 Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013. doi:10.1007/s10009-012-0249-7.
- 37 Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 135–152. ACM, 2013. doi:10.1145/2509578.2509586.
- 38 Darya Verzhbinsky and Daniel Wang. Petsy: Polymorphic enumerative type-guided synthesis. *POPL 2021 Student Research Competition*, 2021.

Modular Abstract Definitional Interpreters for WebAssembly

Katharina Brandl 

Johannes Gutenberg-Universität Mainz, Germany

Sebastian Erdweg 

Johannes Gutenberg-Universität Mainz, Germany

Sven Keidel 

TU Darmstadt, Germany

Nils Hansen

Johannes Gutenberg-Universität Mainz, Germany

Abstract

Even though static analyses can improve performance and secure programs against vulnerabilities, no static whole-program analyses exist for WebAssembly (Wasm) to date. Part of the reason is that Wasm has many complex language concerns, and it is not obvious how to adopt existing analysis frameworks for these features. This paper explores how *abstract definitional interpretation* can be used to develop sophisticated analyses for Wasm and other complex languages efficiently. In particular, we show that the semantics of Wasm can be decomposed into 19 language-independent components that abstract different aspects of Wasm. We have written a highly configurable definitional interpreter for full Wasm 1.0 in 1628 LOC against these components. Analysis developers can instantiate this interpreter with different value and effect abstractions to obtain abstract definitional interpreters that compute inter-procedural control and data-flow information. This way, we develop the first whole-program dead code, constant propagation, and taint analyses for Wasm, each in less than 210 LOC. We evaluate our analyses on 1458 Wasm binaries collected by others in the wild. Our implementation is based on a novel framework for definitional abstract interpretation in Scala that eliminates scalability issues of prior work.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Static Analysis, WebAssembly

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.5

Supplementary Material *Software (Source Code)*: <https://gitlab.rlp.net/plmz/sturdy.scala>
archived at `swh:1:dir:8ccfa27cd16980470eb319533bc03a164d93bf8a`

Funding The German Research Foundation (DFG)–451545561, and ATHENE: National Research Center for Applied Cybersecurity, SeDiTraH

Acknowledgements We thank the anonymous reviewers for their effort and helpful suggestions.

1 Introduction

WebAssembly (Wasm) is a low-level programming language targeted at efficient and portable computation on the web [10]. Wasm modules are often used as a drop-in replacement for computation-intensive JavaScript libraries such as game engines [23, 10]. Wasm has also been designed with security in mind, but many security vulnerabilities reemerge in Wasm because OS-level routines must be provided as user code, which makes them susceptible to attacks [20], and because current compilers targeting Wasm lack protection mechanisms such as stack canaries [29]. While it is well-known that static program analyses can drive performance optimization, reduce binary size, and discover vulnerabilities, no static whole-program analyses exist for Wasm to date.



© Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 5; pp. 5:1–5:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Wasm involves many complex and interacting language features that analyses have to model: operand stacks, call frames, jumps to scoped labels, function and global-variable tables, dynamically loaded modules, and module-owned linear memory to name a few. It is not obvious how to adopt existing analysis frameworks for these features, nor is it obvious how to develop a new analysis framework for these features. In this paper, we demonstrate that *abstract definitional interpretation* is capable of developing analyses for Wasm.

Abstract definitional interpretation was first proposed by Darais et al. [7] as an alternative to abstracting abstract machines [12]. The key idea is to define a generic definitional interpreter that is parametric in value and effect operations, such that it can be instantiated to form concrete as well as abstract interpreters. Keidel et. al. [14] refined this approach to isolate and permit modular reasoning about value and effect components [13]. However, it is unclear if abstract definitional interpretation scales to languages as complex as Wasm and if the resulting analyzers scale to real-world programs of considerable size. In this paper, we answer both of these questions affirmatively and explain how we developed three Wasm analyses in less than 210 LOC each.

The foundation of all our Wasm analyses is a generic definitional interpreter for Wasm, which we designed and implemented. An important contribution of this paper is to decompose the semantics of Wasm and map it to 12 value components and 7 effect components. Each component consists of an interface with a canonical concrete semantics and any number of abstract semantics. Since these components are language-independent, we only have to develop them once and can reuse them across languages and analyses. This way, we managed to develop a fully-fledged definitional interpreter for Wasm 1.0 and its module system in only 1628 lines of language-dependent code.

The generic interpreter is implemented against the interfaces of value and effect components, making the mapping from language concerns to components explicit. Analysis developers can derive abstract definitional Wasm interpreters by selecting an implementation for each component used by the generic interpreter. This makes analysis development modular: We can reuse components between analyses and refine individual components while reusing others unchanged. We demonstrate this modularity by deriving three abstract definitional interpreters from the generic Wasm interpreter: a context-insensitive dead code analysis based on an inter-procedural control-flow graph that we compute, a callsite-sensitive constant propagation analysis, and a callsite-sensitive taint analysis. Each of the three analyses is novel for Wasm, and each of them required less than 210 lines of Wasm-specific code:

	Generic interpreter	Dead code analysis	Constant analysis	Taint analysis
LoC	1628	130	156	209

Technically, our implementation is based on a new framework for definitional abstract interpretation in Scala. Our framework improves over the original DAI by Darais et al. [7] and Sturdy by Keidel et al. [13] to make definitional abstract interpreters scalable. Specifically, our framework exploits a simpler component design and eliminates the monadic transformer stack required by DAI and Sturdy. We show that our analyses scale to real-world programs by analyzing 1458 Wasm binaries collected by others in the wild. Since these binaries are not full applications, we also developed a most general client for Wasm that allows us to apply our whole-program analyses to individual modules soundly. On average, each of our analyses takes 5s per binary, and we find 14% of all instructions are dead code, 10% of all instructions could be replaced by constants, and 56% of all memory accesses are safe against tampering.

	Concrete	Concrete	Type Abs.	Const. Abs.
(func (param i64)	param=1	param=4	param=i64	param=i64
(result i64)	result=1	result=24	result=i64	result=i64
local(i64)				
i64.const 1	[1]	[1]	[i64]	[1]
local.set 1	[]	[]	[]	[]
(loop	[]	[]	[]	[]
local.get 0	[1]	[4]	[i64]	[i64]
i64.const 1	[1,1]	[1,4]	[i64,i64]	[1,i64]
i64.le_u	[1]	[0]	[i32]	[i32]
(if	[]	[]	[]	[]
(then	[]		[]	[]
local.get 1	[1]		[i64]	[i64]
return)	[]		[]	[]
(else		[]	[]	[]
local.get 1		[1]	[i64]	[i64]
local.get 0		[4,1]	[i64,i64]	[i64,i64]
i64.mul		[4]	[i64]	[i64]
local.set 1		[]	[]	[]
local.get 0		[4]	[i64]	[i64]
i64.const 1		[1,4]	[i64,i64]	[1,i64]
i64.sub		[3]	[i64]	[i64]
local.set 0		[]	[]	[]
br 1))))	

■ **Figure 1** Factorial in Wasm: Two concrete runs and an abstract run using a type-based domain.

In summary, we make the following contributions:

- We present the design of a modular analysis platform for Wasm (section 3).
- We decompose Wasm into 12 value components and 7 effect components and implement a generic interpreter against their interfaces (section 4).
- We modularly define 3 whole-program analyses that are novel for Wasm and provide a most general client for Wasm modules (section 5).
- We designed and implemented a new, scalable framework for abstract definitional interpreters in Scala and explain how it improves over prior work. We realized our modular analysis platform for Wasm on top of this framework (section 6).
- We validate the soundness, performance, and applicability of the Wasm analyses (section 7).

2 Introduction to WebAssembly and Problem Statement

Wasm is a low-level stack-based programming language with structured control flow. We illustrate the textual syntax and some of the core features of Wasm using an iterative factorial function in Figure 1 as an example. The leftmost column shows the code of the factorial function, whereas the other columns display the stack of the concrete and abstract executions of that code. Note that the local variable at index 0 refers to the function parameter and is used as an iteration counter, whereas the local variable at index 1 is an accumulator for the result of the factorial function.

We illustrate the concrete interpretation of the factorial function for arguments 1 and 4. Most Wasm operations interact with the operand stack whose contents we show in Figure 1 for each instruction. For example, `i64.const` and `local.get` push values to the stack, whereas `local.set` and `i64.le_u` pop values from the stack. For `param=1`, the `if` finds that the argument is less-equal than 1 and thus terminates. For `param=4`, the `if` goes to the else-branch, where we accumulate the factorial result, decrement the iteration counter, and jump to the beginning

of the loop. Jumps in Wasm are structured, which means they can only target enclosing blocks, indexed by distance. In our example, `br 1` jumps over the `if`-block and targets the loop. After a few more iterations, we will again reach the `then`-branch where the loop terminates.

To illustrate the abstract interpretation of Wasm, the two rightmost columns in Figure 1 show an abstract evaluation of the factorial function where values are approximated by their types and by concrete values if they are constant. The factorial function is called with type `i64` as argument, denoting any 64-bit integer. Each abstract evaluation must overapproximate both concrete evaluations. Hence the abstract interpreter analyzes both branches of the `if`-instruction and loop until reaching a fixed point. This type analysis can be used to derive a control-flow graph, but the value representation is configurable in our system. Later in this paper, we present Wasm analyses that use more precise value abstractions.

Wasm provides many other interesting features not shown in our illustrating example. For instance, in addition to normal function calls, there are also indirect function calls whose call target can be found in a function table. Functions can also be imported from other modules and Wasm code can invoke external functions provided by the runtime system. When Wasm runs in the browser, these external functions are JavaScript programs. Finally, each Wasm module can declare module-global variables and request a linear memory (i.e., a byte array) to store data.

Problem Statement

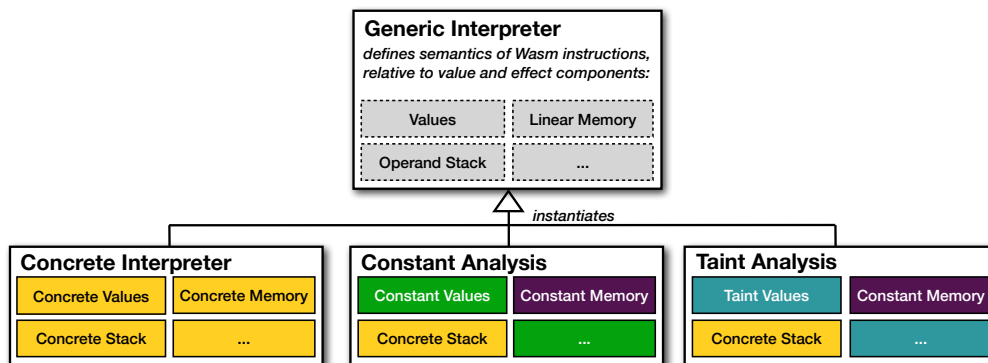
We want to develop abstract interpreters for Wasm that track data-flow and information-flow. This is a difficult challenge since the abstract interpreter has to deal with all of Wasm's concerns: the operand stack, call frames, global variables, linear memory, function tables, and structured jumps. Without modularity, all concerns have to be handled at once, complicating the initial development and hindering evolution.

For example, consider the semantics of indirect function calls which combines 5 Wasm concerns highlighted with italic font: The interpreter first pops the *numeric index* of the function from *operand stack* and uses it to search through the *function table* to find the function definition. If the table has a function definition of the correct type at the index, the interpreter invokes the function. In particular, the interpreter binds the function arguments on the operand stack to the function parameters on a newly created *call frame*. Finally, the interpreter processes the body of the function and afterwards pushes the return argument on the stack. There are also multiple edge cases which cause the function invocation to *fail*.

A naive monolithic analysis implementation may closely couple the semantics of indirect calls to specific abstractions for the function index, the operand stack, call frame, and failures. This coupling not only complicates the analysis implementation, it also makes it difficult to change the abstractions without also requiring changes to the abstract semantics of indirect calls. To solve this problem, we divide and conquer by modularizing the analysis implementation, which we discuss in the following section.

3 Modular Wasm Analyses in a Nutshell

In this section, we present the design of our modular analysis platform for Wasm. At the core of our platform is a generic definitional interpreter for Wasm. The generic interpreter describes the semantics of Wasm instructions and serves as a template to derive different Wasm analyses, as well as a concrete interpreter. The generic interpreter is parametric in its representation for values such as integers and floating point values. Furthermore, the generic interpreter is parametric in its representation of effects such as the linear memory or the



■ **Figure 2** We propose a modular Wasm analysis platform with a generic interpreter at its root.

```

trait GenericInterpreter[V, ExcV]:
  // Independent value components for abstract value type V
  val i32ops: IntegerOps[Int, V]
  val f64ops: FloatOps[Double, V]
  // Independent effect components
  val stack: OperandStack[V]
  type WasmExc[V] = (JumpTarget, List[V])
  val except: Except[WasmExc[V], ExcV]
  // Interpreter written against value and effect components
  def evalInst(inst: Inst): Unit = inst match
    case i32.Sub =>
      val v2 = stack.popOrFail(); val v1 = stack.popOrFail()
      stack.push(i32ops.sub(v1, v2))
    case f64.Abs =>
      val v = stack.popOrFail()
      stack.push(f64ops.abs(v))
    case Return =>
      val operands = stack.popNOrFail(currentReturnArity)
      except.throws((JumpTarget.Return, operands))

```

■ **Figure 3** Simplified generic interpreter that handles subtraction, absolutes, and function returns.

operand stack. Analyses instantiate the generic interpreter with different abstractions for values such as constants, taint flags, or types and with different abstraction for effects such as a constant memory abstraction. Similarly, the concrete interpreter instantiates the generic interpreter with concrete values and effects.

Our platform is modular along two dimensions. First, the generic interpreter defines the semantics for Wasm instructions once and for all; analyses simply reuse that semantics. Second, the values and effects required by the generic interpreter are decomposed into language-independent components, which can be defined language-independently and reused flexibly. Figure 2 illustrates the modularity of our platform. The generic interpreter sits on top and is instantiated to obtain concrete and abstract interpreters. It depends on various value and effect components that must be provided during instantiation. In Figure 2, the colors illustrate component reuse. While each interpreter uses a different value representation, the two abstract interpreters use the same component for linear memory and the operand stack. Since the shape of the operand stack is decidable in Wasm [10], this component is also shared with the concrete interpreter. In the remainder of this section, we illustrate how our analysis platform realizes the generic interpreter, its instances, and the components.

Generic interpreter

Figure 3 shows a simplified generic interpreter for Wasm. The generic interpreter does not refer to any specific concrete or abstract value representations. Instead, the interpreter abstracts over them with the value components `IntegerOps` for 32-bit integers and `FloatOps` for 64-bit floats. Value components are interfaces with any number of implementations, for example:

```

trait IntegerOps[B, V]:           // a type class for integer operations
  def integerLit(i: B): V         // - embeds base literals of type B into the value type V
  def sub(v1: V, v2: V): V       // - subtraction of two values
object ConcreteIntegerOps extends IntegerOps[Int, Int] {...}           // concrete semantics
object ConstantIntegerOps extends IntegerOps[Int, Topped[Int]] {...} // constant abstraction
object SignLongIntegerOps extends IntegerOps[Long, Sign] {...}       // sign abstraction

```

In addition to the value components, the simplified generic interpreter requests two components for effects: one for the mutable operand stack and one for exception handling. Like value components, effect components define an interface that can be implemented in various ways. The `OperandStack[V]` effect component provides `push`, `pop`, and `peek` operation for values of type `v`. The `Except` component provides operations for throwing and catching exceptions of type `WasmExc[V]`, consisting of a jump target and a list of operand values. In contrast to prior frameworks for abstract definitional interpretation, we distinguish value from effect components to improve the run-time performance of our analyses. Specifically, value components capture pure operations and do not contribute to the analysis state, whereas effect components maintain internal state that is part of the overall analysis state. This becomes relevant when joining computations or computing the fixpoint of an analysis.

The generic interpreter only relies on the interfaces of value and effect components. Based on these, the generic interpreter defines the semantics of Wasm instructions with the interpretation function `evalInst`. We only show a few selected cases. For integer subtraction, function `evalInst` pops two values from the stack, subtracts them, and pushes the result back on the stack. Note that most Wasm instructions are not overloaded, so it is easy to select the appropriate value component. For example, function `evalInst` delegates the instruction `f64.Abs` to the component `f64ops`, which handles 64-bit floating-point numbers. The operand stack is ubiquitous in the generic interpreter, but other effects are needed too. For example, function `evalInst` implements return instructions using exceptions that are caught at the function head. Exception handling is a standard way for implementing non-local control flow on the JVM, where our analyzers run. Exception handling also closely aligns with jumps and returns in Wasm: Due to the structured control flow of Wasm, all jumps (including returns) target a surrounding block. Similarly, exceptions interrupt execution and return to the closest surrounding exception handler.

Concrete interpreter

We can instantiate the generic interpreter for different value and effect components. In particular, we can derive a concrete Wasm interpreter by choosing the canonical concrete semantics for all components and lifting them to Wasm values. Specifically, we represent Wasm values using the corresponding number types of the JVM, because the definitional Wasm interpreter is implemented in Scala.

```

enum Value:
  case I32(i: Int); case I64(l: Long); case F32(f: Float); case F64(d: Double)

```

With this, we can instantiate the generic interpreter:

```
class ConcreteInterpreter extends GenericInterpreter[Value, WasmExc[Value]]:
  val i32ops = ... // lifts IntegerOps[Int, Int] to Value.I32
  val f64ops = ... // lifts FloatOps[Double, Double] to Value.F64
  val stack = new ConcreteOperandStack[Value]
  val except = new ConcreteExcept[WasmExc[Value]]
```

For values we lift the canonical concrete semantics to the `Value` type, for effects we select all required effect components directly from our library.

Abstract interpreter

We can derive abstract interpreters in the same manner. For example, let us build a type analysis that only distinguishes the type of each value:

```
enum Type:
  case I32; case I64; case F32; case F64; case Top
```

Wasm does not need `Top`, but we include it so `Type` forms a semi-lattice. We instantiate the generic interpreter using `Type` for values and join exceptions that jump to the same target:

```
type ExcByTarget = Map[JumpTarget, List[Type]]
class AbstractInterpreter extends GenericInterpreter[Type, ExcByTarget]:
  val i32ops = // lifts IntegerOps[Int, IntType] to Type.I32
  val f64ops = // lifts FloatOps[Double, DoubleType] to Type.F64
  val stack = new JoinableConcreteOperandStack[Type]
  val except = new JoinedExcept[WasmExc[Type], ExcByTarget]
```

Our platform provides language-independent type abstractions for various components. For the value components in Wasm, we lift these abstractions to the Wasm-specific abstraction `Type`. For the operand stack, we exploit that its shape is decidable for Wasm, which allows us to reuse the concrete operand stack (through subclassing). The abstract interpreter must join the contents of stacks at control-flow join points, but these stacks will have equal size. For exceptions, we select an abstract semantics that collects all possibly active exceptions in a set. Although not shown here, analyses can select a context-sensitivity and configure other aspects of the fixpoint algorithm, such as the iteration strategy or loop unrolling depth.

This example illustrates how our platform supports the modular development of Wasm analyses: by plugging together value and effect components and instantiating the generic interpreter. Moreover, individual components can be refined and replaced easily. But how can we decompose Wasm into value and effect components and define a generic interpreter for the full language?

4 Decomposing Language Concerns of WebAssembly

In this section, we propose a decomposition of Wasm that separates individual language concerns into components. We will then define a Wasm generic interpreter on top of these components. The generic interpreter only uses the interfaces of the components, while concrete and abstract interpreters instantiate the generic interpreter with selected implementations of the components. This way, the decomposition of Wasm into components enables analysis developers to compose full-fledged Wasm analyses modularly.

In the remainder of this section, we present our decomposition of Wasm and its mapping to value and effect components. For each component, we have implemented the canonical concrete semantics compatible with the Wasm specification. We show possible abstract semantics in section 5, where we construct data and information-flow analyses for Wasm.

4.1 Values

Wasm defines four different value types, namely integers and floats with 32 and 64 bits: `i32`, `i64`, `f32`, `f64`. In section 3, we already showed how some of the value components can be used to implement value operations generically, such as `IntegerOps` for implementing operations on integers. However, we omitted many details for illustration purpose. The goal of this subsection is to fill the gap and to introduce other value components we used for Wasm. Throughout this section, the type variable `v` stands for the abstract value type used by the generic interpreter.

Numeric operations

We decompose the numeric operations of Wasm into 6 value components. Besides components for the various arithmetic operations of the four value types, we use one component for equality testing, and one component for ordering comparisons of Wasm values:

```
val i32ops: IntegerOps[Int, V]           val f32ops: FloatOps[Float, V]
val i64ops: IntegerOps[Long, V]        val f64ops: FloatOps[Double, V]
val eqOps: EqOps[V, V]                 val orderingOps: OrderingOps[V, V]
```

The mapping from Wasm instructions to the respective components is straightforward, but it is not a one-to-one mapping; some instructions combine multiple operations from components:

```
def evalIntegerUnaryOperation(op: IUnop, v: V): V = op match
  case i64.Extend32S =>
    val shift = i64ops.integerLit(32)
    i64ops.shiftRight(i64ops.shiftLeft(v, shift), shift)
```

Also note that the validation of Wasm rejects comparisons on values of different type. Thus, when providing instances for `EqOps` and `OrderingOps`, it is sufficient to consider those cases where the operands have the same type.

Conversions

Wasm features many operations that convert between value types. For example, there are three operations converting from `i32` values to `f32` values, namely signed and unsigned conversions and byte reinterpretation. We use a single `Convert` interface for all conversions, but require 12 different instances of that component:

```
trait Convert[From, To, VFrom, VTo, Config]:
  def apply(from: VFrom, conf: Config): VTo

val convert_i32_i64: Convert[Int, Long, V, V, ..]
val convert_i32_f32: Convert[Int, Float, V, V, ..]
val convert_i32_f64: Convert[Int, Double, V, V, ..]
...
```

Note that the first two type parameters `From` and `To` of `Convert` are tags or phantom types: They are only used to describe the component. The actual values to be converted are of type `VFrom` and `VTo`, both of which we instantiate with `v` in the generic interpreter. Actual instances consider specific value representations for `VFrom` and `VTo`, and we lift these instances to operate on values `v` as described below. The `Config` parameter guides the conversion. For example, the following code handles the three different conversions of `i32` to `f32` values:

```
def evalConvertop(op: Convertop, v: V): V = op match
  case f32.ConvertSI32 => convert_i32_f32(v, Signed)
  case f32.ConvertUI32 => convert_i32_f32(v, Unsigned)
  case f32.ReinterpretI32 => convert_i32_f32(v, Raw)
```

The `Convert` interface can not only be used for numeric conversion operations. We use the same interface for operations that serialize and deserialize values into bytes. This is required to write values into Wasm's linear byte memory:

```
val encode: Convert[V, Seq[Byte], V, Bytes, ...]
val decode: Convert[Seq[Byte], V, Bytes, V, ...]

def evalInst(inst: Inst): Unit = inst match
  case i: StoreInst =>
    val v = stack.popOrFail()
    val bytes = encode(v, ...)
    ... // store bytes in memory
```

Branching

Concrete and abstract interpreters differ significantly when it comes to branching control flow, as required for conditional constructs. While the concrete interpreter will select exactly one branch to execute, abstract interpreters must analyze both branches unless they can statically decide if the branching condition is true or false. We capture branching with a value component that receives two continuations:

```
trait BoolBranching[B, R]:
  def boolBranch(v: B, thn: => R, els: => R): R
```

Implementations of this interface can select the type `B`, for which they can decide the branching. For example, we show the canonical concrete semantics that instantiates `B` with `Boolean` and a type semantics that uses `BooleanType`:

```
class ConcreteBranch[R] extends BoolBranching[Boolean, R]:
  def boolBranch(v: Boolean, thn: => R, els: => R): R = if (v) thn else els

class BoolTypeBranch[R](eff: EffectStack, j: Join[R]) extends BoolBranching[BooleanType, R]:
  def boolBranch(v: BooleanType, thn: => R, els: => R): R = eff.joinComputations(thn, els, j)
```

The concrete semantics simply uses the boolean condition to decide which branch to execute. In contrast, the type semantics must execute both branches and join their results and effects. Our platform provides a helper function `joinComputations` to achieve that, given the stack of effects (`EffectStack`) used by the abstract interpreter and an instance of type class `Join[R]`. In our implementation, these arguments are modeled as implicit parameters and resolved automatically. We explain how our framework joins effectful computations in section 6.

We use `boolBranch` for all conditional instructions: `select`, `brif`, and `if`. For example:

```
val branchOps: BooleanBranching[V, Unit]
def evalInst(inst: Inst): Unit = inst match
  case If(bt, thnInsts, elsInsts) =>
    val isZero = evalNumeric(i32.Eqz)
    branchOps.boolBranch(isZero, label(elsInsts), label(thnInsts))
```

We will explain the `label` function later in the context of jumps. For now it is sufficient to know that it executes a labeled block of code.

Lifting Value Components

Our platform provides language-independent concrete and abstract instances for all value components, such as the concrete `IntegerOps[Int, Int]` and the abstract `IntegerOps[Int, IntType]`. However, as shown above, generic interpreters usually require operations on some compound

5:10 Modular Abstract Definitional Interpreters for WebAssembly

type for values. To reuse the language-independent component instances, we must lift them to the Wasm-specific value type. To facilitate this, our platform provides lifting instances for all value components, which can be easily instantiated. For example, the following two definitions lift the concrete and type-based integer operations to Wasm values and types, respectively:

```
val i32opsValue: IntegerOps[Int, Value] =
  new LiftIntegerOps({case Value.I32(i) => i}, i => Value.I32(i))
val i32opsType: IntegerOps[Int, Type] =
  new LiftIntegerOps({case Type.I32 => IntType}, _ => Type.I32)
```

For an underlying value type U , `LiftIntegerOps` takes an extract function $v \Rightarrow U$ and an inject function $U \Rightarrow v$. With these, it wraps the operations of the underlying language-independent component instance, for example:

```
def sub(v1: V, v2: V): V = inject(underlying.sub(extract(v1), extract(v2)))
```

In our Wasm analyses, all value components are based on language-independent component instances that we lift.

4.2 Effects

Computations generally yield values and trigger effects. Wasm features many language concerns that are effectful. We capture these concerns in effect components. While value components are stateless, effect components contain internal state. This distinction is important when joining computations (as in the type-based `boolBranch`), because effect components must participate in the join (see section 6 for details). In this subsection, we present a decomposition of Wasm's effectful language concerns into effect components.

Operand Stack

Wasm programs interact with an operand stack. We capture this effect in a dedicated effect component:

```
trait OperandStack[V, MayJoin[_]]:
  def push(v: V): Unit
  def pop(): JOption[MayJoin, V]
  def popOrFail(): V = ...
  ...
```

Except for the `MayJoin` type parameter, this component provides a standard stack interface. The `MayJoin` parameter determines whether the component can yield an uncertain result for `pop`. For example, if an abstract stack semantics lost track of the stack's height, `pop` would yield an uncertain result that comprises alternative values or even a stack underflow. In contrast, a concrete stack semantics yields certain results only: either the stack's topmost value or no value if the stack is empty. Instances of `OperandStack` can declare which behavior they provide by choosing `NoJoin` or `WithJoin` for `MayJoin`:

```
enum MayJoin[A]:
  case NoJoin()
  case WithJoin(j: Join[A], eff: EffectStack)
```

Indeed, a concrete stack uses `NoJoin` whereas an abstract stack uses `WithJoin`. Given a `WithJoin[A]`, we can invoke `joinComputations` as shown above in the abstract branching semantics of subsection 4.1. Furthermore, `Join[A]` is used to join values of type `A`. `OperandStack` forwards the `MayJoin` parameter to `JOption`, a data type for joinable option values that we use to represent uncertain data. Since `JOption[NoJoin, A]` is isomorphic to the standard `Option[A]`, concrete operand stacks provide a standard stack interface.

Many of our effect components use a similar design to declare that operations may yield uncertain results in the abstract semantics. Indeed, the generic interpreter itself has a `MayJoin` parameter that it forwards to the required effect components. However, sometimes the generic interpreter can formulate more precise requirements. For Wasm, the language specification guarantees that the height of the stack is decidable at all times and that stack lookups must yield certain results. To this end, the generic Wasm interpreter requires a decidable operand stack, which internally selects `NoJoin` for `MayJoin`.

Indirect Calls and Function Tables

Wasm features indirect function calls via function indices, which really are plain `i32` values computed by the program. To evaluate an indirect function call, Wasm reads a function index from the stack, looks up the index in a function table, and invokes the found function:

```
def evalInst(inst: Inst): Unit = inst match
  case CallIndirect(typeIx) =>
    val funcIx = stack.popOrElseFail()
    val funV = funTable.getOrElse(funcIx, fail(UnboundFunctionIndex, ...))
    funOps.invokeFun(funV, invoke)
```

This code uses two additional components: an effect component `funTable` and a value component `funOps`. We model the function table as a generic `SymbolTable` component that maps symbols to entries:

```
trait SymbolTable[Symbol, V, MayJoin[_]]:
  def get(symbol: Symbol): JOption[MayJoin, V]
  def put(symbol: Symbol, newEntry: V): JOption[MayJoin, Unit]

val funTable: SymbolTable[FuncIx, FunV, MayJoin]
```

Note how the symbol table uses the same `MayJoin` pattern as the operand stack. However, lookups in the function table are not decidable in Wasm, so that abstract interpreters sometimes obtain an uncertain function. For example, our type analysis does not track the values of function indices and thus must consider all reachable functions as potential targets for indirect calls. This also is the reason why the function table contains `FunV` values rather than functions directly: We must be able to join function values. To abstract from the specific `FunV` representation, we use a generic value component `FunctionOps`:

```
trait FunctionOps[Fun, A, R, FunV]:
  def funValue(fun: Fun): FunV
  def invokeFun(v: FunV, a: A)(invoke: (Fun, A) => R): R

val funOps: FunctionOps[Function, FuncType, Unit, FunV]
```

Operation `funValue` lifts a function into a function value `FunV`. Operation `invokeFun` does the inverse: It extracts functions from a function value and applies the continuation `invoke` on each of them. Similar to `boolBranch`, abstract instances of `FunctionOps` join the result `R` of all functions.

Global Variables

Wasm features numerically indexed global variables that can be used to store values. We model global variables using the same `SymbolTable` component that we used for function tables. However, the resolution of global variables is decidable in Wasm and always yields a certain result. We incorporate this fact in the generic interpreter by requiring a decidable symbol table for global variables:

```
val globals: DecidableSymbolTable[Int, V]
```

5:12 Modular Abstract Definitional Interpreters for WebAssembly

Please note that in Wasm, each module has its own globals, function table, and memory, which can also be shared between modules. Our implementation takes this into account, but we decided to simplify the presentation of the code for the paper.

Local Variables

Each Wasm function can declare local variables, which we understand to include the function parameters. A function can read and write its local variables freely. We model local variables through a generic `CallFrame` component. Each call frame has a fixed size determined at construction by operation `inNewFrame`. In addition, a call frame can track auxiliary `Data` for each frame. For Wasm, we use the call frame to track the module instance of the currently executing function as well as its return arity:

```
trait CallFrame[Data, Var, V, MayJoin[_]]:
  def inNewFrame[A](d: Data, vs: Seq[(Var, V)])(f: => A): A
  def getFrameData: Data
  def getLocal(x: Int): JOption[MayJoin, V]
  def setLocal(x: Int, v: V): JOption[MayJoin, Unit]

val callFrame: DecidableCallFrame[(ModuleInst, Int), Int, V]
```

Note how both call frames and symbol tables map indices to values. However, call frames are scoped by function call and the previous call frame is restored when exiting a function. Operation `inNewFrame` takes care of this behavior, executing `f` in the new frame and restoring the previous frame after `f` finishes. This way, the generic interpreter can implement function invocations:

```
def invoke(fun: Function): Unit =
  val args = stack.popNOrFail(fun.params.size)
  val locals = args ++ fun.locals.map(num.defaultValue)
  val data = (module, fun.returnArity)
  callFrame.inNewFrame(data, locals)(enterFunction(fun))
```

Linear Memory

Wasm programs can load and store data from a growable linear memory. Technically, the linear memory is a byte array that is accessed using 32-bit integers as index. Wasm provides various instructions to load and store values of different types. In our generic interpreter, the following code handles load instructions using the memory effect component:

```
trait Memory[Addr, Bytes, Size, MayJoin[_]]:
  def read(addr: Addr, length: Int): JOption[MayJoin, Bytes]
  def write(addr: Addr, bytes: Bytes): JOption[MayJoin, Unit]

val memory: Memory[Addr, Bytes, Size, MayJoin]
def load(inst: LoadInst): Unit =
  val addr = effectiveAddr(inst.offset)
  val length = getBytesToRead(inst)
  val bytes = memory.read(addr, length).orElse(fail(MemoryAccessOutOfBounds, ...))
  stack.push(decode(bytes, inst))
```

We first compute the effective address to be loaded by adding a static offset to the base address, which is on the operand stack. We then determine the number of bytes to be loaded. We invoke the read operation of the memory effect component to obtain a byte sequence. Finally, we decode those bytes using the `decode` component discussed in subsection 4.1.

Jumps

Wasm features a limited form of jumps that abides by structured control flow, which means that jumps can only target enclosing blocks. Instead of using named labels, Wasm jumps declare the number of blocks to skip, that is, the block-distance between the jump and the target block. We model jumps through an effect component for exception handling:

```
trait Except[Exc, ExcV, MayJoin[_]]:
  def throws(ex: Exc): Nothing
  def tries[A](f: => A): JEither[MayJoin, A, ExcV]
```

The `Except` component is parametric in the underlying exception type `Exc` and the representation of exception values `ExcV`. Similar to `JOption` from above, operation `tries` yields a value of a joinable either data type, `JEither` for short. That is, `tries` either yields an `A` when `f` triggers no exception, or it yields an `ExcV`. Since abstract instances of `Except` may not be able to determine the exact behavior of `f`, the result of `tries` can be uncertain, which `JEither` encapsulates.

The generic interpreter uses exception handling to support jumps and returns:

```
type WasmExc[V] = (JumpTarget, List[V])
enum JumpTarget:
  case Jump(labelIndex: LabelIdx)
  case Return

val except: Except[WasmExc[V], ExcV, MayJoin]

def jump(labelIndex: LabelIdx): Unit =
  val returnArity: Int = labelStack.arityOf(labelIndex)
  val operands = stack.popNOrFail(returnArity)
  except.throws((JumpTarget.Jump(labelIndex), operands))

def label(returnArity: Int, insts: Seq[Inst]): Unit =
  labelStack.pushLabel(returnArity)
  val tried = except.tries(insts.foreach(evalInst))
  labelStack.popLabel()
  tried.either(identity) {
    case (JumpTarget.Jump(0), ops) => stack.pushN(ops)
    case (JumpTarget.Jump(ix), ops) => except.throws(WasmExc.Jump(ix - 1, ops))
    case (JumpTarget.Return, ops) => except.throws(WasmExc.Return(ops))
  }
```

Function `jump` takes the index of a label, looks up the return arity required by that label in an auxiliary data structure called `labelStack`, and triggers a `Jump` exception with the corresponding number of operands. `Jump` exceptions are handled by function `label`, which we use when entering a new block. This function first pushes the return arity of the label to the `labelStack` and then tries to run all instructions of the block. We use `either` to react to the result of that execution. If the block succeeds without exception, nothing has to be done (`identity`). However, if an exception was (possibly) thrown, we react accordingly. If the jump target has index 0, it targets the current label and we push the operands on the stack. Otherwise, we decrement the jump target index and escalate the exception. Return exceptions always escalate; they are handled by `enterFunction`.

Traps

Wasm programs can trigger unrecoverable errors, called traps. We model traps using the `Failure` effect.

```

trait Failure:
  def fail(kind: FailureKind, msg: String): Nothing

val failure: Failure

```

In contrast to exceptions, failures are unrecoverable and cannot be caught. While the canonical concrete semantics of `Failure` aborts the execution of a Wasm program, abstract interpreters must continue to explore execution paths that do not fail. That is, the abstract `fail` produces a set of potential `FailureKind` and throws a specific Scala failure exception. Furthermore, the failure join operation catches failure exceptions at branching points and continues to explore other branches. After all branches have been explored, the failure join operation rethrows the failure exception if one of the branches failed.

4.3 Summary

We have decomposed the analysis of Wasm into various language concerns. We implemented each of these concerns with 12 separate value components for numeric operations, conversions, and branching, and with 7 effect components for the operand stack, function and symbol tables, global and local variables, linear memory, jumps, and traps. Based on this decomposition, we have developed a generic interpreter for Wasm that is parametric in how the value and effect components are instantiated. The generic interpreter implements evaluation of Wasm code. The generic interpreter also implements the module system, manages exports, resolves imports, and performs module instantiation, which is used to initialize variables, function tables, and memories. In particular, we have implemented the canonical concrete semantics for all value and effect components and used those to derive a concrete Wasm interpreter. This concrete Wasm interpreter is a feature-complete and correct implementation of the Wasm 1.0 specification, as we detail in section 7.

The generic interpreter is not only parametric in the value and effect components, but also in the fixpoint algorithm. While the concrete interpreter can simply run a program until it terminates, abstract interpreters must widen analysis results to ensure termination. To this end, our generic interpreter is written in an open recursive style, giving control to the fixpoint algorithm in each recursive invocation. When instantiating the generic interpreter, we configure a generic fixpoint algorithm provided by our platform to select context-sensitivity and other aspects. We illustrate such configuration in the next section, where we build three whole-program Wasm analyses as instances of the generic interpreter.

5 Modularly Defined Analyses for Wasm

In the previous section, we have presented the key ingredients of our modular static analysis platform for Wasm: a Wasm semantics decomposed into value and effect components and a generic Wasm interpreter. In the present section, we demonstrate how our platform can be used to implement Wasm analyses modularly. To this end, we implement three Wasm analyses: a dead code analysis, a constant propagation analysis, and a taint analysis. We compose each analysis modularly from value and effect components that we use to instantiate the generic interpreter.

5.1 Type Analysis

As a baseline, we first describe an analysis with a type abstraction, which additionally identifies dead code. To this end, we must construct an inter-procedural control-flow graph (CFG) that allows us to identify unreachable instructions. Note that the construction of a precise interprocedural CFG is undecidable in general and approximation is required. In this subsection, we use a type analysis to approximate the behavior of the program.

Our platform provides a reusable singleton type `BaseType[T]` to represent type `T`, which we use to model our type analysis:

```
enum Type:
  case I32(i: BaseType[Int]); case I64(l: BaseType[Long]);
  case F32(f: BaseType[Float]); case F64(d: BaseType[Double]); case Top

type Addr = BaseType[Int]
type Bytes = BaseType[Seq[Byte]]
type Size = BaseType[Int]

type FuncIx = BaseType[Int]
type FunV = Powerset[FunctionInstance]
type ExcV = Map[JumpTarget, List[Type]]
```

The type analysis does not track memory access precisely: all reads yield a top value. Specifically, we represent addresses `Addr`, byte sequences `Bytes`, and memory size `Size` using their type. We also don't track function indices: Indirect function calls resolve to the set of all functions currently in the function table. For exceptions, we collect all active exceptions in a set. Based on these definitions, we select the following effect components:

```
val stack = new JoinableConcreteOperandStack[Type]
val memory = new TopMemory[MemoryAddr, Addr, Bytes, Size]
val globals = new JoinableConcreteSymbolTable[GlobalAddr, Type]
val funTable = new UpperBoundSymbolTable[TableAddr, FuncIx, FunV]
val callFrame = new JoinableConcreteCallFrame[FrameData, Int, Type]
val except = new JoinedExcept[WasmException[Value], ExcV]
val failure = new AFailureCollect
```

Note how we use decidable instances for the operand stack, call frames, and global variables, since all three concerns are statically decidable in Wasm. The memory yields top on every read, the function table yields all stored entries when queried. We use the `AFailureCollect` instance for abstract failures, which collects all possible failures of the analyzed program.

Finally, every analysis must configure the fixpoint algorithm used by our platform. Most importantly, we must select a context-sensitivity and iteration strategy. Our platform provides a combinator library for describing these aspects:

```
val phi = fix.log(controlFlowGraphLogger,
  fix.contextSensitive(fix.context.none,
    fix.filter(isFunOrLoop, fix.iter.innermost))
```

Combinator `fix.contextSensitive` determines the context-sensitivity of the type analysis. Specifically, the type analysis is context-insensitive, which means that all calls of the same function are joined. Combinator `fix.filter` applies the inner combinator only to instructions for which predicate holds. In this case, the filter combinator applies a specific iteration strategy to functions and loops, because these are the only Wasm constructs which can diverge and need to be iterated on. Combinator `fix.iter.innermost` iterates on the innermost strongly-connected components of the dependency graph of the abstract interpreter. Specifically, it iterates on the innermost of nested loops and the innermost of nested recursive function calls. Lastly, combinator `fix.log` calls a logger before and after every instruction. The logger in this case records an interprocedural control-flow graph, which we explain in the following paragraph.

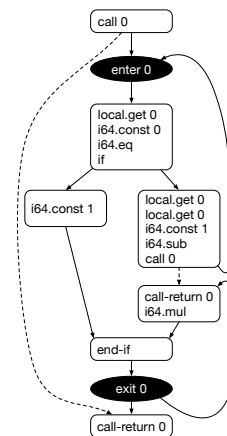
CFG construction

Our platform uses big-step abstract interpretation, in which the control flow of analyzed programs is implicit. However, we can make the control flow explicit by observing the order in which instructions are executed by the abstract interpreter. To this end, we call function `fix.control` of our platform with mappings from Wasm instructions to CFG nodes:

```

val controlFlowGraphLogger = fix.control(config) {
  // called before interpreting an instruction
  case Enter(fun) => CfgEnter(fun)
  case Eval(c: Call, loc) => CfgCall(c, loc)
  case Eval(inst, loc) => CfgInstruction(inst, loc)
} {
  // called after interpreting an instruction
  case (Enter(fun), Exit(_)) => CfgExit(fun)
  case (Eval(c: Call, loc), _) => CfgCallReturn(c, loc)
}

```



Function `fix.control` returns a logger, that is called before and after each Wasm instruction. The logger adds instructions to basic blocks, adds control-flow edges between basic blocks, and adds call edges between call-site, entry, and exit points of functions.

For example, this code constructs the CFG shown on the right for a recursive factorial function, where dashed lines represent call-return edges. Of course, the CFG construction also scales to larger examples. The last line in the code above activates CFG logging for a given `analysis`. While our type analysis is context-insensitive, other analyses may exploit context-sensitive CFGs. But, as we show in section 7, even the simple type analysis already produces useful results and finds dead code in Wasm programs. Furthermore, the CFG can be used as a starting point for other analysis approaches.

5.2 Constant Propagation Analysis

We define a constant propagation analysis by refining the type analysis from above. In a constant propagation analysis, values are either a concrete value or `Top`:

```

enum Value:
  case I32(i: Topped[Int]);   case I64(l: Topped[Long]);
  case F32(f: Topped[Float]); case F64(d: Topped[Double]); case Top

type Addr = Topped[Int]
type Bytes = Seq[Topped[Byte]]
type Size = Topped[Int]

type FuncIx = Topped[Int]
type FunV = Powerset[FunctionInstance]
type ExcV = Map[JumpTarget, List[Type]]

```

Notably, the constant propagation analysis tracks constant memory addresses and bytes. That is, when writing a concrete value to a known address, we store the concrete byte encoding of the value. Conversely, when reading from a known address, if we find a concrete byte sequence, we decode it into a concrete value. This memory abstraction is certainly only a first step in developing sophisticated Wasm analyses, but our modular analysis platform allows us to refine it in future work. For function indices, we track their precise index if possible. Ideally, dereferencing a function index yields a single function that we can execute, but if the function index is `Top`, we obtain a set of all functions in the function table.

Compared to the type analysis, we only have to adapt two effect components, namely those that handle memory and function indices. We highlight the differences in blue font:

```

val stack = new JoinableConcreteOperandStack[Type]
val memory = new ConstantAddressMemory[MemoryAddr, Addr, Bytes, Size]
val globals = new JoinableConcreteSymbolTable[GlobalAddr, Type]
val funTable = new ConstantSymbolTable[TableAddr, FuncIx, FunV]
val callFrame = new JoinableConcreteCallFrame[FrameData, Int, Type]
val except = new JoinedExcept[WasmException[Value], ExcV]
val failure = new AFailureCollect

```

To increase the precision of the constant propagation analysis, we can choose a 1-callsite sensitive fixpoint algorithm. To this end, we log each function call with a call-site logger and use the most recent call site as a context:

```
val callSites = fix.context.callSites {
  case Eval(c: (Call | CallIndirect), _) => Some(c)
  case _ => None
}
val phi = fix.log(callSites,
  fix.log(controlFlowGraphLogger,
    fix.contextSensitive(callSites.callString(1),
      fix.filter(isFunOrLoop, fix.iter.innermost))))
```

Finally, we need to determine whether an instruction is constant in all execution paths. We can achieve this by observing the results of the abstract interpreter for each instruction. To this end, we implemented a logger that reads the relevant data from the operand stack before and after executing an instruction. In case an instruction is visited more than once (e.g., in a loop) the recorded values are joined. If the final result is constant, the instruction is constant across all execution paths. Our analysis platform allows us to add this functionality modularly:

```
val constants = new InstructionLogger { inst =>
  // log before execution of inst
  if (readsSingleValueFromStack(inst))
    Some(stack.peekOrFail())
  else if ...
} { inst =>
  // log after execution of inst
  if (writesSingleValueToStack(inst))
    Some(stack.peekOrFail())
  else if ...
}
```

5.3 Taint Analysis

As a last example, we define a taint analysis by refining the constant propagation analysis. The goal of the analysis is to detect tainted memory accesses, i.e., if a tainted value is used as memory address. As source for tainted values, we consider user input which results from calling host functions. To track taint, we tag a taint property to each value:

```
enum Value:
  case I32(i: Taint[Topped[Int]]); case I64(l: Taint[Topped[Long]]);
  case F32(f: Taint[Topped[Float]]); case F64(d: Taint[Topped[Double]]); case Top

type Addr = Topped[Int]           type FuncIx = Topped[Int]
type Bytes = Seq[Taint[Topped[Byte]]] type FunV = Topped[Powerset[FunctionInstance]]
type Size = Topped[Int]           type ExcV = Map[JumpTarget, List[Type]]
```

We omit the effect and fixpoint configuration of the taint analysis since it is identical to the constant propagation analysis.

To detect illegal memory access through tainted values, we add a new observer to the analysis. Note that we observe the values on the stack before they are cast to an address, which is why type `Addr` does not need a taint flag.

```
val tainting = new InstructionLogger { inst =>
  if (isLoadInst(inst)) {
    val addrV = stack.peekOrFail()
    if (addrV.isTainted) Some(Powerset(addrV)) else None
  }
}
```

We collect tainted addresses for each memory instruction. A memory instruction is safe if its set of tainted addresses is empty. Of course, we could track other sinks or sources for tainted values and expect to do so in future work.

5.4 Most General Client for Wasm Modules

Abstract definitional interpreters are whole-program analyses: Interpretation starts in the main function and subsequently explores all code reachable from there. However, Wasm programs are usually used as libraries within JavaScript applications. To apply our whole-program analyses to individual Wasm modules, we develop a most general client for Wasm.

Most general clients can be used to apply whole-program static analyses to library code [19]. A most general client approximates all valid usages of a given library, and it can be used as a single entry point for the analysis. We have developed a most general client for Wasm modules that exercises all interleavings of all exported functions in a loop:

```
def runMostGeneralClientLoop(modInst: ModuleInstance): Unit =
  effectStack.mapJoin(modInst.exportedFunctions) { case (funName, funIx) =>
    val fun = modInst.functions.getOrElse(funIx, fail(UnboundFunctionIndex, funIx.toString))
    val args = fun.funcType.params.map(typedTop).toList
    invokeExported(modInst, funName, args)
  }
  fixpoint(runMostGeneralClientLoop(modInst))
```

In each loop iteration, we run all exported functions in isolation and join their effects to update the analysis state. Our fixpoint algorithm iterates this loop until the analysis state is stable. The final analysis state soundly approximates all possible sequences of exported functions.

Note that a Wasm client can also write to exported tables and memory. Our most general client does not capture this behavior, which may cause the analysis result to be unsound for such clients. If the exported tables and memory are not edited externally, our approach obtains a sound analysis result for the library code.

6 A Scalable Framework for Abstract Definitional Interpretation

We designed and implemented a new framework for abstract definitional interpretation in Scala as open source.¹ In this section, we describe how our new framework improves over prior work and why that was necessary for scaling the approach to complex languages and real-world programs. There are two prior frameworks for abstract definitional interpretation: the original DAI in Racket by Darais et al. [7] and Sturdy in Haskell by Keidel et al. [13]. While we compare to both, we also implemented a complete generic definitional interpreter for Wasm in Sturdy and report on the lessons learned.

Component design

Abstract definitional interpretation has supported modularly defined components from the start. Already in DAI, the generic PCF interpreter used components for environments, stores, and allocation [7]. However, these components followed an ad-hoc design and did not share an interface between concrete and abstract semantics. Not only did this preclude modular reasoning about components, it also implies that we must use the non-determinism

¹ <https://gitlab.rlp.net/plmz/sturdy.scala>

monad to collect alternative analysis (sub-)results. For example, DAI features a function `isZero(v: V): Boolean` in the concrete semantics and `isZero(v: V): List[Boolean]` in the abstract semantics. Consequently, when the abstract semantics cannot decide if a value is zero it yields `List(true, false)` and all of the remaining analysis is run twice: once for `true` and once for `false`. Nested conditionals with uncertain conditions like this trigger an exponential blow-up that is unacceptable when scaling up.

Sturdy was designed to support the development of sound static analyses with compositional soundness proofs. For this reason, Sturdy introduced a design principle based on parametricity that ensures no details about the concrete or abstract semantics is leaked into the generic interpreter [14]. This design principle prohibits an operation `isZero` as in DAI. Instead, Sturdy provides a operation `ifZero(v: V, ifTrue: => R, ifFalse: => R): R`, where `ifTrue` and `ifFalse` are continuations. If both continuations must be run, Sturdy joins their results before moving on with the rest of the analysis. Sturdy uses a similar design for all operations that introduce uncertainty. For example, reading from a store is done by operation `read(a: Addr, ifFound: V => R, ifNotFound: => R): R`. We found the use of continuations in Sturdy excessive, making it harder to write and maintain the generic interpreter for Wasm. But can this be avoided?

In our framework, we have retained Sturdy’s design principles to permit modular reasoning about components. While our framework does not attempt to support formal proofs, modular reasoning reemerges in the form of modular soundness propositions that can be used during testing. However, we significantly reduce the amount of continuations needed by encapsulating uncertain results in dedicated auxiliary data types: `JOption` and `JEither`. These data types provide standard operations such as `getOrElse`, `map`, and `flatMap`. For the concrete semantics, these data types behave identical to the standard `Option` and `Either` types, but their abstract semantics can encode uncertainty such as `LeftOrRight(1, r)`. Besides reducing the number of continuations needed, these types significantly improve the readability of component interfaces. For example, reading from a store has the simple signature `read(a: Addr): JOption[MayJoin, V]`.

Eliminating the monadic transformer stack

Both DAI and Sturdy encode the generic interpreter in monadic style: The side effects triggered by the analyzed program are threaded through the monadic computation. And both frameworks use transformers to decompose effect handling into components. For example, in Figure 4 we show the transformer stacks used by DAI and Sturdy for a k-CFA analysis of PCF, as well as the transformer stack for our prototypical constant propagation analysis of Wasm implemented in Sturdy. This shows how the transformer stack grows considerably when analyzing complex languages.

Large transformer stacks are problematic because they impair the performance of the interpreter. Every monadic operation in the interpreter must traverse the entire transformer stack, slowing down interpretation considerably. Keidel et al. [13] measured this effect and showed that an interpreter on a transformer stack was 7756x slower than the same computation after exhaustive inlining of the entire stack. Thus, they argued that inlining allows us to enjoy modularity without regrets. While we concur in principle, this approach does not scale to complex languages unfortunately. For transformers stacks like the one for Wasm shown in Figure 4, the compiler exceeded 16 GB of memory while inlining and ultimately failed to compile the program. Since a 7756x slower analysis is not feasible, we must find an alternative design to support modularly defined components.

In our framework, we follow an object-oriented design in representing independent components. Rather than stacking all components and threading their effect through the computation, we let each component manage and manipulate its own internal state. As

5:20 Modular Abstract Definitional Interpreters for WebAssembly

```
// DAI: k-CFA analysis of PCF, 6 components
ReaderT (FailT (StateT (NondetT (CacheT (FinMap0 Power0) ID))))

// Sturdy: k-CFA analysis of PCF, 8 components
ValueT (ErrorT (EnvT (FixT (ComponentT (StackT (CacheT (CallSiteT (->))))))))

// Sturdy: constant propagation of Wasm, 15 components
ValueT (JumpTypesT (OperandStackT (ExceptT (StaticGlobalStateT
  (MemoryT (SerializeT (TableT (FrameT (LogErrorT
    (FixT (ComponentT (StackT (CacheT (ControlFlowT (->))))))))))))))
```

■ **Figure 4** Deep transformer stacks as required by DAI and Sturdy impair the performance of the analyzers.

usual in OO, the internal state is encapsulated in the component and hidden behind a public interface. For example, setting a global variable `globals.set(x, stack.popOrFail())` changes the internal state of `stack` and `globals`, which is observable through operations of the public interface, such as `globals.get`. Since components are not stacked, invoking a component's operation is a simple method call that does not involve any other components.

Only when joining effectful computations, all effect components must participate, each taking care of their own internal state. The generic interpreter defines an effect stack that determines the order in which effects are joined. For Wasm, we use the following effect stack:

```
val effectStack = EffectStack(List(
  stack, memory, globals, funTable, callFrame, except, failure))
```

Each abstract semantics of an effect component must implement `joinComputations(f)(g)`, which executes `f` and `g` on the current internal state and merges the two resulting states. We apply a common strategy to implement these joins:

1. Take a snapshot of the internal state.
2. Execute `f`, store the resulting state.
3. Restore the snapshot state.
4. Execute `g`, store the resulting state.
5. Join the two states in an effect-dependent manner.

Consider the following example program:

```
// locals before: 0 := 0; 1 := 10
(if (then (i32.const 25) (local.set 0)) (else (local.get 0) (local.set 1)))
// locals after: 0 := (25 ⊔ 0); 1 := (10 ⊔ 0)
```

The `then` branch produces a call frame that still maps `0 := 25` and `1 := 10` unchanged. The `else` branch must operate on a copy of the original call frame and produce `0 := 0` unchanged and `1 := 0`, ignoring the manipulations done in the `then` branch. Finally, we join the resulting call frames, obtaining the result shown above. In the next section, we show that analyses defined in our framework scale to real-world programs.

7 Evaluation

section 5 has already demonstrated how our approach enables the modular construction of Wasm analyses. In this section, we present empirical results that attest (i) the concrete interpreter is correct, (ii) the static analyses are sound with respect to the concrete interpreter, and (iii) the type, constant, and taint analyses yield relevant results.

Correctness of concrete interpreter

Establishing the correctness of the concrete interpreter is important, because the concrete interpreter provides ground truth for reasoning about the soundness of our analyses. Thus, any soundness result we may provide is only meaningful as long as the concrete interpreter itself is a true implementation of the Wasm specification. In particular, our analyses and the concrete interpreter share the generic interpreter, which must be correct. In fact, if there was a bug in the generic interpreter, this bug would *not* trigger a soundness violation, because the concrete interpreter would exhibit the same incorrect behavior. Therefore, establishing the correctness of the concrete interpreter is paramount.

To this end, we ran our concrete Wasm interpreter against the official test suite from the Wasm specification.² The test suite consists of 16481 assertions, testing the correct behaviour of the Wasm interpreter. This testing revealed several bugs in our implementation, all of which we fixed. For example, we found indexing errors in the linear memory and several subtle bugs concerning floating-point operations. Our concrete Wasm interpreter now passes the complete test suite.

Soundness of static analyses

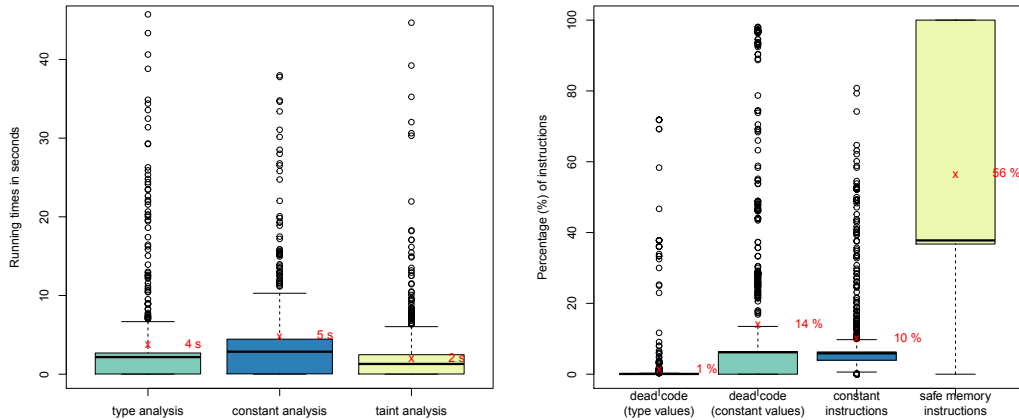
Only sound analyses can be used to inform program optimizations without jeopardizing the program's semantics. Since we want to conduct performance optimizations and reduce the size of Wasm binaries, we must ensure our analyses are sound. To this end, we tested soundness of our analyses against the concrete interpreter. Our platform allows us to implement soundness propositions for each value and effect component modularly. Value components implement an abstraction function that lifts the canonical concrete value representation into the abstract domain, using a partial order on the abstract domain to determine sound approximation. Effect components implement a soundness proposition that relates the internal state of the canonical effect implementation to their own internal state. That is, we not only check the final value computed by an analysis, but also the final state of the linear memory and other effect components. An analysis then simply composes the soundness propositions of its components.

We tested the soundness of our analyses against the concrete interpreter on the test suite from the Wasm specification. Specifically, we ran the analyses and the concrete interpreter simultaneously and tested analysis soundness after every single assertion. This uncovered several bugs. For example, we initially defined integer division $\text{Top} / \text{Top} = \text{Top}$, which neglects division-by-zero errors and should yield $\text{Top} \sqcup \text{fail}(\dots)$ instead. We were able to fix all soundness bugs, so that we are confident the abstract interpreters are sound with respect to the concrete interpreter.

Large-scale evaluation

To assess the applicability and performance of our analyses, we applied them to the programs collected by others in the WasmBench benchmark suite. WasmBench [11] contains 8461 unique Wasm binaries collected from various sources, including github, NPM, and by crawling websites. Out of these, we had to ignore 7003 binaries that failed to validate, 6354 of which due to unresolvable imports of modules not collected by the benchmark suite. Since WasmBench collects individual binaries rather than applications, we have no principled

² <https://github.com/WebAssembly/spec/>



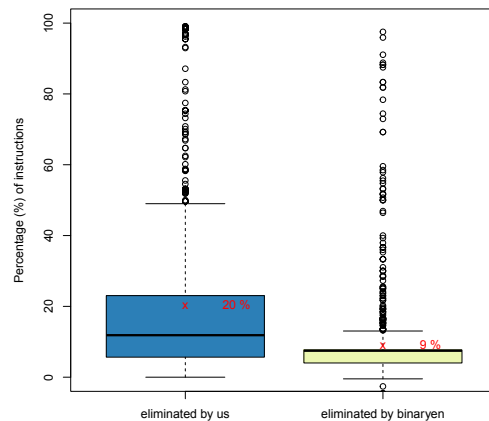
■ **Figure 5** Running times in seconds (left) and analysis results in % of instructions (right) for analyzing each of the 1458 WasmBench binaries. The red cross indicates the mean time or percentage.

means of finding the right module. Another 607 binaries out of the 7003 were rejected due to invalid memory page size information. For each binary of the remaining 1458 binaries, we run our analyses using the most general client described in subsection 5.4, so that the analysis results soundly approximate any potential usage of the module.

We measured the running times after a warm-up phase. We cancelled analysis runs after 60 seconds, which yielded between 196 and 200 timeouts per analysis. This timeout was chosen for pragmatic reasons: To limit the overall time required to run the experiment, which finishes in a little over 7 hours. Figure 5 shows the running times of the successful analysis runs. On average, the type analysis finishes in 4s, the constant analysis in 5s, and the taint analysis in 2s. The taint analysis is faster because it does not construct a call graph. We note that 81% of all type and constant analysis runs finish in 10s or less (including those runs that timed out), as do 85% of all taint analysis runs.

Figure 5 shows the percentage of instructions our type-based dead code, constant-based dead code, constant propagation analysis, and taint analysis identified. We count an instruction as dead if it is unreachable or, in case of blocks and loops, if they are never targeted by a jump. Such dead instructions can be safely eliminated from a Wasm binary. This reduces the binary size and saves bandwidth if the binary is sent over the network. Unsurprisingly, our baseline type analysis cannot find much dead code. However, even a simple constant propagation analysis can already reduce binaries by 14% on average. Note that the dead code this analysis identified was missed by other compilers, as many of the binaries stem from deployed packages and websites. The constant analysis also identifies 10% of instructions as computing constant results. This excludes instructions like `i32.const` of course. Constant instructions can be replaced by such `const` instructions. Due to our modular architecture, analysis developers can focus on improving one aspect of the analysis at a time to increase the optimization potential further.

Finally, the goal of the taint analysis is to track the data flow of tainted values and detect if tainted values can reach critical program points. Our taint analysis defines user input and results of calling host functions as tainted and detects potential security risks if tainted values are used as memory addresses. Protecting the memory is important because many compilation schemes targeting Wasm use the memory to embed critical infrastructure of the source language’s runtime system [20]. For example, some runtime systems manage their own call stack in the memory, which thus is not protected from the user. If we can show that



■ **Figure 6** Comparing our approach to Binaryen, the industry standard for Wasm optimizations.

the user cannot access or and manipulate the memory shape, this means that the runtime system cannot be tampered with this way. Consequently, we consider a memory access to be safe if the analysis can guarantee that a tainted (user-influenced) value cannot be used as an address. On average, our analysis finds 56% of all memory accesses to be safe. Out of the 1458 Wasm binaries, our analysis shows 28% to be completely safe, meaning they only contain safe memory accesses. This analysis is fairly simple still and, for example, does not support any sanitization of tainted values, which should further improve the analysis results.

Comparison with the industry standard

While we compare to related work in the subsequent section, we thought it is important to validate our approach empirically in comparison to the industry standard. The de-facto industry standard for Wasm code optimization is Binaryen³, a C++ library that provides its own Wasm IR and implements about 100 optimization passes in its `wasm-opt` tool. This includes whole-program constant propagation and dead code optimizations, although the details and limits of the underlying analyses are not clearly documented. This begs the question: Can our approach compete with Binaryen, an industry standard for Wasm optimization developed by more than 140 contributors.

We answer this question quantitatively by running the optimizer of Binaryen on all WasmBench binaries that we successfully optimized. Binaryen transforms the Wasm code into its own IR, optimizes that IR, and translates it back into the Wasm binary format. We configured Binaryen using the `-oz` flag, which aggressively optimizes for code size. We compute the number of *eliminated instructions* by loading the original and the optimized module and subtracting their instruction counts. We then compare this number to our constant analysis, where each dead or constant instruction counts toward the *eliminated instructions*. Figure 6 shows the results of our experiment.

Our experiment clearly shows that our approach outperforms Binaryen in terms of precision, eliminating twice as many instructions on average. While further investigation is necessary to understand where exactly our approach wins compared to Binaryen, note that

³ <https://github.com/WebAssembly/binaryen>

we have built a generic framework for Wasm analyses. In particular, constant propagation is a simple abstract domain and we may expect far better precision by using intervals or even relational abstract domains. Our framework is designed to accommodate those future improvements. In terms of performance, Binaryen only takes 0.1s on average, where our callsite-sensitive constant propagation analysis takes 4.8s on average. This is to be expected, given that our analysis lies in a different complexity class.

One important threat to validity of this experiment is that our analyses do not actually rewrite Wasm binaries. Instead, we count the number of instructions that were detected as dead or constant. We believe this is fair, since dead instructions can be dropped for sure and the constant instructions can be removed by propagating the constant value. Actually, we penalizes our own approach because in `i32.const 1; i32.const 2; i32.add`, we only count the last instruction as eliminable, while Binaryen removes all three of them. We hope to integrate our analysis into a framework like Binaryen in future work to realize optimizations based on our analysis results.

8 Related Work

Our work investigates how to develop modular static analyses for Wasm using abstract definitional interpreters. We have already compared to prior approaches of abstract definitional interpreters in section 6 in detail. In this section, we discuss how our work relates to prior work on Wasm, x86 assembly, and JVM bytecode.

Stiévenart and Roover [28] designed the first static taint analysis *Wassail* for Wasm using a compositional approach. In particular, they analyze each function in isolation and compute a summary of the taint information of the following form:

```
function 8: stack: [10,11], globals: [g0;11], mem: g7
```

This example summary means that the Wasm function with id 8 may store the variables 10, 11 on stack, may store the variables g0, 11 as globals, and variable g7 in the linear memory. In a second step, they combine the summaries of multiple functions in bottom-up order of the call graph to compute the complete analysis result. While compositional analyses are known to scale better, they are also less precise than whole-program analyses. There are two places where our whole-program taint analysis is more precise than *Wassail*'s compositional taint analysis. First, *Wassail* does not resolve indirect calls precisely. In particular, an indirect call reads the function index from the stack, which is not approximated by *Wassail*. Instead, *Wassail* resolves an indirect call to all functions which have a matching type [2]. This may be especially imprecise for common function signatures such as `F64 -> F64`. In contrast, our constant taint analysis approximates the stack and is able to resolve indirect calls precisely in case the function index is a constant. Second, *Wassail* does not approximate the layout of Wasm's linear memory precisely. In particular, *Wassail* returns all taint variables stored in memory on every load instruction. In contrast, our constant taint analysis approximates the layout of Wasm's linear memory more precisely. Specifically, we have distinct read behavior for constant addresses and top addresses. Reading from a top address yields the memories upper bound, which is the default behavior for all reads in *wassail*, but constant addresses result in actual lookups. This increases the precision of load instructions with a constant address.

Wasp⁴ is a C++ library for performing simple static analyses on Wasm code. It offers methods to dump specific parts of a module (e.g., all functions) and to compute a function's call graph, control-flow graph, and data-flow graph. In contrast to our work, Wasp is not

⁴ <https://github.com/WebAssembly/wasp>

designed to implement more sophisticated analyses for Wasm but rather as a tool making it easy to work with Wasm modules. In particular, Wasp does not consider abstract domains to approximate values and thus, by and large, yields results equivalent to our type analysis. But, as our evaluation showed, even simple value domains such as constant propagation improve the precision of analyses significantly: The type analysis only found 1% of dead instructions on average, whereas we were able to prove 14% of instructions are dead using an abstract domain for constant propagation. This is out of reach for Wasp.

Wasabi [21] is a general purpose framework for implementing *dynamic* analyses for Wasm, which can be implemented using a high-level JavaScript API. The framework then instruments the Wasm binary to call these JavaScript analysis functions. Dynamic analyses are used in different contexts than static analyses. While analyses for security (e.g., a taint analyses) may be performed both statically and dynamically, compiler optimizations entail the use of a static analysis. Hence, the focus of their work is orthogonal to ours and explores a different part of the design space.

Watt et al. [34] developed two formal semantics for Wasm in the Isabelle and Coq proof assistants. These formal semantics can be used to prove properties about Wasm programs. However, these proofs require a high amount of manual effort and expertise in contrast to static analyses, which are automatized.

Static analysis of x86 assembly code [3, 6, 16] faces several challenges summarized in the PhD thesis of Kinder [15]. For example, unstructured control-flow with `goto`'s and long jumps with dynamic jump target complicate the construction of a control-flow graph [17, 24]. Furthermore, x86 programs store their code alongside the data during the execution, which makes it harder for static analyses to differentiate between them [33]. This also allows x86 programs to modify their own code during execution, which poses a severe challenge for static analyses [30]. In contrast, Wasm prevents these problems with a stricter language design. In particular, Wasm is statically-typed, features only structured control-flow and clearly separates between code and data, which makes it impossible for Wasm programs to modify their own code [10]. The stricter language design of Wasm lowers the bar for implementing static analyses and improves their precision compared to x86 analyses.

Many static analysis frameworks for Java target JVM bytecode [8, 4, 27], the assembly code that underlies the Java Virtual Machine [22]. However, JVM bytecode poses a challenge to static analyses, because of its implicit dataflow and due to the use of a stack. Vallee-rai and Hendren [32] solved this problem by compiling JVM byte code to *Jimple*, a simpler three-address code. Jimple is easier to analyze than JVM bytecode, because the addresses relieve from having to extract dataflow information from the stack. Since its inception, Jimple has become the defacto standard for analyzing JVM bytecode and is used by popular Java analysis frameworks such as Doop [25, 9] and Soot [31, 5, 1, 26]. In contrast, we show that abstract definitional interpretation can be used to analyze Wasm code directly, without requiring another intermediate representation, such as Jimple. This is a key advantage of abstract definitional interpretation.

Koren [18] presented an integrated development environment for Wasm that can be used to develop high-performance and latency-sensitive Wasm applications for the internet of things. Such an IDE would benefit from static analyses built with our modular platform, as static analyses can provide valuable feedback to the developer about low-level and hard to understand Wasm programs.

Lehmann et al. [20] and Stiévenart et al. [29] investigated the security risk of compiled Wasm programs. In particular, C applications compiled to Wasm reexperience security problems that are well known and fixed in the native C compiler. More specifically, the compiled C programs are vulnerable to stack and heap-based buffer overflow attacks. These vulnerabilities can be detected by static analyses for Wasm code.

9 Conclusion

In this work, we developed the first whole-program control and data-flow analyses for Wasm based on abstract interpretation. It is important that we understand how to analyze Wasm programs for enabling optimizations and to find bugs and vulnerabilities. Our analyses lay the foundation for that as they scale to real-world programs, where we find 14% of all Wasm instructions are dead code, 10% of all instructions can be replaced by constants, and 56% of all memory accesses are safe against tampering.

Our analyzers are based on two core contributions this paper makes. First, we present a decomposition of the Wasm semantics into 19 language-independent components that abstract different aspects of Wasm. This decomposition allowed us to develop static analyses modularly, which was essential for limiting the complexity of the implementation and the development effort. Second, we show how abstract definitional interpretation can be used to implement modularly defined static analyses for complex languages at scale. We explained how our new framework for abstract definitional interpretation eliminates the inefficiencies of prior frameworks, and why that was crucial for scaling to complex languages and real-world programs. The lessons learned for building abstract definitional Wasm interpreters can certainly be transferred.

References


- 1 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Oceau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O’Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom – June 09 – 11, 2014*, pages 259–269. ACM, 2014. doi:10.1145/2594291.2594299.
- 2 Darren C. Atkinson. Accurate call graph extraction of programs with function pointers using type signatures. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November – 3 December 2004, Busan, Korea*, pages 326–335. IEEE Computer Society, 2004. doi:10.1109/APSEC.2004.16.
- 3 Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 – April 2, 2004, Proceedings*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer, 2004. doi:10.1007/978-3-540-24723-4_2.
- 4 Roberto Barbuti, Nicoletta De Francesco, and Luca Tesei. An abstract interpretation approach for enhancing the java bytecode verifier. *Comput. J.*, 53(6):679–700, 2010. doi:10.1093/comjnl/bxp031.
- 5 Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and soot. In Eric Bodden, Laurie J. Hendren, Patrick Lam, and Elena Sherman, editors, *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, pages 3–8. ACM, 2012. doi:10.1145/2259051.2259052.
- 6 Marco Cova, Viktoria Felmetzger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC’06)*, pages 269–278. IEEE, 2006.
- 7 David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *PACMPL*, 1(ICFP):12:1–12:25, 2017.
- 8 Julian Dolby, Stephen J Fink, and Manu Sridharan. Watson libraries for analysis (wala). URL: <http://wala.sf.net/>.
- 9 Neville Grech and Yannis Smaragdakis. P/taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):102:1–102:28, 2017. doi:10.1145/3133926.

- 10 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.
- 11 Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *WWW: The Web Conference*, pages 2696–2708. ACM / IW3C2, 2021.
- 12 David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 51–62. ACM, 2010.
- 13 Sven Keidel and Sebastian Erdweg. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360602.
- 14 Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. Compositional soundness proofs of abstract interpreters. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–26, 2018.
- 15 Johannes Kinder. *Static analysis of x86 executables (Statische Analyse von Programmen in x86-Maschinensprache)*. PhD thesis, Darmstadt University of Technology, 2010. URL: <http://tuprints.ulb.tu-darmstadt.de/2338/>.
- 16 Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 2008. doi:10.1007/978-3-540-70545-1_40.
- 17 Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2009. doi:10.1007/978-3-540-93900-9_19.
- 18 István Koren. A standalone webassembly development environment for the internet of things. In Marco Brambilla, Richard Chbeir, Flavius Frasinca, and Ioana Manolescu, editors, *Web Engineering*, pages 353–360. Cham, 2021. Springer International Publishing.
- 19 Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for javascript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 83–93. IEEE / ACM, 2019.
- 20 Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann>.
- 21 Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 1045–1058. ACM, 2019. doi:10.1145/3297858.3304068.
- 22 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
- 23 Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment – 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, volume 11543 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2019. doi:10.1007/978-3-030-22038-9_2.

- 24 Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan, and Mizuhito Ogawa. A hybrid approach for control flow graph construction from binary code. In Pornsiri Muenchaisri and Gregg Rothermel, editors, *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 – Volume 2*, pages 159–164. IEEE Computer Society, 2013. doi:10.1109/APSEC.2013.132.
- 25 Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, 2015. doi:10.1561/2500000014.
- 26 Johannes Späth, Karim Ali, and Eric Bodden. *Ide^{al}*: efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA):99:1–99:27, 2017. doi:10.1145/3133923.
- 27 Fausto Spoto. The julia static analyzer for java. In Xavier Rival, editor, *Static Analysis – 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2016. doi:10.1007/978-3-662-53413-7_3.
- 28 Quentin Stiévenart and Coen De Roover. Compositional information flow analysis for webassembly programs. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 – October 2, 2020*, pages 13–24. IEEE, 2020. doi:10.1109/SCAM51674.2020.00007.
- 29 Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. The security risk of lacking compiler protection in webassembly, 2021. arXiv:2111.01421.
- 30 Tayssir Touili and Xin Ye. Reachability analysis of self modifying code. In *22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017*, pages 120–127. IEEE Computer Society, 2017. doi:10.1109/ICECCS.2017.19.
- 31 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot – a java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999. URL: <https://dl.acm.org/citation.cfm?id=782008>.
- 32 Raja Vallee-rai and Laurie Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.
- 33 Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Differentiating code from data in x86 binaries. In Dimitrios Gunopulos, Thomas Hofmann, Donato Malerba, and Michalis Vazirgiannis, editors, *Machine Learning and Knowledge Discovery in Databases – European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part III*, volume 6913 of *Lecture Notes in Computer Science*, pages 522–536. Springer, 2011. doi:10.1007/978-3-642-23808-6_34.
- 34 Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two Mechanisations of WebAssembly 1.0. In *FM 2021 – Formal Methods*, pages 1–19, Beijing, China, November 2021. URL: <https://hal.archives-ouvertes.fr/hal-03353748>.

Dynamically Updatable Multiparty Session Protocols

Generating Concurrent Go Code from Unbounded Protocols

David Castro-Perez ✉ 

University of Kent, UK

Nobuko Yoshida ✉ 

University of Oxford, UK

Abstract

Multiparty Session Types (MPST) are a typing disciplines that guarantee the absence of deadlocks and communication errors in concurrent and distributed systems. However, existing MPST frameworks do not support protocols with *dynamic unbounded participants*, and cannot express many common programming patterns that require the introduction of new participants into a protocol. This poses a barrier for the adoption of MPST in languages that favour the creation of new participants (processes, lightweight threads, etc) that communicate via message passing, such as Go or Erlang.

This paper proposes *Dynamically Updatable Multiparty Session Protocols*, a new MPST theory (DMst) that supports protocols with an *unbounded* number of fresh participants, whose communication topologies are *dynamically updatable*. We prove that DMst guarantees deadlock-freedom and liveness. We implement a toolchain, GoScr (Go-Scribble), which generates Go implementations from DMst, ensuring **by construction**, that the different participants will only perform I/O actions that comply with a given protocol specification. We evaluate our toolchain by (1) implementing representative parallel and concurrent algorithms from existing benchmarks, textbooks and literature; (2) showing that GoScr does not introduce significant overheads compared to a naive implementation, for computationally expensive benchmarks; and (3) building three realistic protocols (dynamic task delegation, recursive Domain Name System, and a parallel Min-Max strategy) in GoScr that could not be represented with previous theories of session types.

2012 ACM Subject Classification Theory of computation → Program specifications; Computing methodologies → Concurrent programming languages

Keywords and phrases Multiparty Session Types, Correctness-by-construction, Concurrency, Golang

Digital Object Identifier 10.4230/LIPICs.ECOOP.2023.6

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*:
<https://doi.org/10.4230/DARTS.9.2.10>

Funding This work is supported by EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, EP/T014512/1, and Horizon EU TaRDIS 101093006.

Acknowledgements We deeply thank Benito Echarren Serrano for his initial collaboration on a preliminary version of this work.

1 Introduction

Multiparty Session Types. Multiparty Session Types (MPST) are typing disciplines that can guarantee the absence of deadlocks and communication errors in concurrent and distributed systems [21, 22]. MPST allow the specification of *global communication protocols (global types)* among a number of participants. The **projection** operation extracts the *local communication protocols (local types)*, from the point of view of each participant in the



© David Castro-Perez and Nobuko Yoshida;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 6; pp. 6:1–6:30



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



system. Projection only succeeds when the protocol is absent of deadlocks and communication errors. These local types can then be used to typecheck processes [21], generate **correct by construction** code [25, 3], or monitor to detect communication errors at runtime [8].

However, MPST have a severe limitation: they cannot model protocols in which *new participants join the system*. Many important protocols rely on this. For example, Chord [54] is a popular protocol for distributed hash tables where participants join a ring, and relies on a stabilisation protocol to guarantee that each participant keeps up-to-date channels to their successors and predecessors. To model such scenarios using MPST, it would be necessary to *interleave* different sessions. But arbitrary session interleavings can lead to deadlocks, so it must be restricted [2, 5]. This not only rules out the use of MPST for many realistic scenarios, but also limits the applicability of MPST for languages that favour process creation and message passing, such as **Go**, which is the main motivation of our work.

Dynamic (Unbounded) Participants in Go. Go is a concurrent programming language designed in 2009 by Google, and it is increasingly popular among professional developers. According to a 2020 Stack Overflow survey, Go is used by 9.4% of developers, and it is the “third most wanted language” [52]. Go was also the 4th most active language in GitHub in 2020 [16], and it has been adopted in many large software systems such as Kubernetes [32], gRPC [18] and Docker [13]. Its main features are explicit communication primitives, namely *channels* and *goroutines* (lightweight threads), whose design comes from concurrent process calculi [20, 40, 41]. Unfortunately, a recent empirical study reveals that over 50% of Go concurrent bugs are caused by communication [56, 39, 61] (i.e., more than shared memory bugs). While Go includes a global *runtime* deadlock detector, it is neither adequate to verify applications with complex communication structures, nor can it detect deadlocks involving only a strict subset of a program’s goroutines (partial deadlocks) [37].

Figure 1 illustrates Go’s core concurrency constructs. It shows a server (Master) that processes client requests (Line 4), and sends responses back to the Client (Line 20). The Master breaks down the request into different subtasks and delegates them to different Worker goroutines (Lines 7–10). The Master then aggregates the Worker results (Lines 11–19). If the Master receives an error message, it will forward it to the Client and stop processing any new messages (Lines 16–19). This program uses a common Go computation pattern¹, the master-worker pattern, and the number of workers depends on a **runtime value**.

Unfortunately, there is a bug in the implementation in Figure 1. The implementation uses synchronous channels. Since the Master goroutine stops processing Worker responses after receiving the first error message, all other goroutines which have not sent their result or error messages will be **deadlocked**, as they will be stuck waiting for the Master to process their message. One might think that this error could be fixed by replacing the synchronous channels in the implementation with asynchronous (buffered) channels. Unfortunately, this approach leaves *orphan messages* which could introduce other concurrency bugs, e.g. the Master may need to clean up resources after receiving a response from the Workers.

This example demonstrates how even in simple programs, message passing can introduce concurrency bugs and channel leakage, violating **deadlock-freedom** and **liveness**. While, in simple programs, these concurrency bugs can be fixed with relative ease, identifying and fixing them is usually done during testing phase, which becomes increasingly harder as the complexity of the program and the number of goroutines increases. Unfortunately, standard MPST cannot model protocols such as Figure 1, since the number of participants is not fixed at the start, and depends on a run-time value.

¹ E.g. https://github.com/tmrts/go-patterns/blob/master/messaging/fan_out.md

```

1 func Worker(n int, resp chan int, err chan error) { ... } // Worker returns either result or error
2 func Master(reqCh chan int, respCh chan []int, cErrCh chan error) {
3   for {
4     ubound := <-reqCh // Receive request from Client
5     workerChs := make([]chan int, ubound) // Array to store worker result channels
6     errCh := make(chan error)
7     for i := 0; i < ubound; i++ { // n_workers depends on runtime value
8       workerChs[i] = make(chan int) // Create worker channel
9       go Worker(i+1, workerChs[i], errCh)
10    }
11    var res []int
12    for i := 0; i < ubound; i++ { // Aggregate worker results
13      select {
14        case sql := <-workerChs[i]: // Aggregate successful result
15          res = append(res, sql)
16        case err := <-errCh: // Some worker failed
17          cErrCh <- err // Propagate error and
18            return // stop processing any further messages
19      }
20    }
    respCh <- res // Send final result to client
  }
}

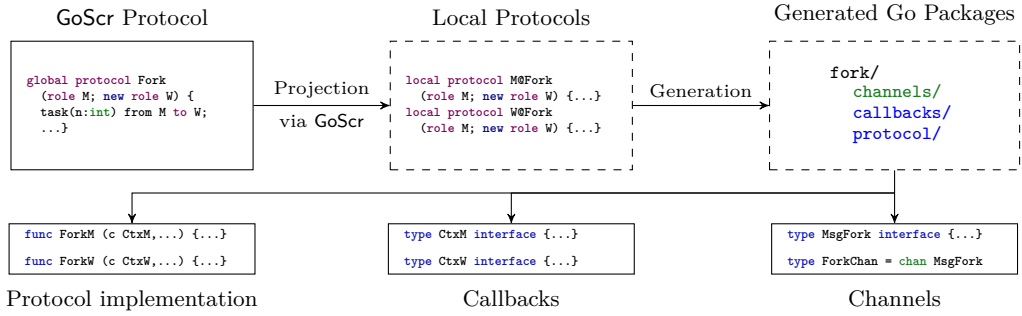
```

■ **Figure 1** Dynamic task delegation implementation in Go (unsafe).

Adding dynamic participants to MPST. This paper introduces *Dynamically Updatable Multiparty Session Types* (DMst), a new theory of MPST whose **novel feature** is to model protocols in which participants can join an already existing session (*dynamic participants*). DMst can guarantee *deadlock-freedom* and *liveness* (*partial-deadlock-freedom*) **by construction** in such protocols. We implement DMst as a tool, GoScr, which generates **correct by construction Go code**, and we evaluate it on a number of representative algorithms in Go, including a safe version of Figure 1 (see § 5.2(a)). While our target language is Go, DMst is not Go specific and a part of GoScr (GoScr protocols, projection and local protocols in Figure 2) is reusable for any language, as long as it supports (1) the creation of new participants (threads or processes) and (2) communication between participants.

Contributions. DMst overcomes several bottlenecks of existing theories on session types (A); and the two main lines of work (B,C) for static deadlock detection in Go:

(A) Dynamic Participants and Session Types. There are two main existing theoretical lines of work related to dynamic MPST. *Dynamic Multirole Session Types* (MRST) [9] enable a set of participants which belong to the same group (i.e. role) to join a multiparty session type. The roles are *fixed at the start*, and can only join at specific points in the protocol, e.g at the beginning of each protocol iteration. *Nested MPST* [7] model protocols with unbounded new participants. Neither MRST nor Nested MPST can represent DMst protocols (A-1) where participants join dynamically to recursive protocols, except at fixed points and with fixed roles (see Example 6). In addition, (A-2) our theory provides stronger guarantees than [7], while their global types are more complex, as they must be checked by a complex typing system. Hence a safe version of Figure 1 cannot be represented by [7, 9]. Both of [9, 7] are *only theoretical*, and lack any implementation or practical results. DMst’s global types are not only more expressive than those in [9, 7], but also simpler, thus DMst is more suitable for real language implementations. Other lines of work add session types to calculi that allow dynamic participants, or extend MPST to specify where can participants join in a protocol, e.g. [19, 57, 58, 30]. While these lines of work can add or replace participants to a system, these participants must act according to known, fixed roles. Therefore, these lines of work do not allow the specification of cyclic recursive topologies that change dynamically with the introduction of new participants.



■ **Figure 2** Overview of GoScr toolchain.

(B) Inference Approach. This approach verifies safety and liveness properties of Go programs, by using model-checking on their *inferred* concurrent behavioural types [47, 36, 37, 14]. The major limitations of this approach are: **(B-1)** there is a gap between properties of types and programs, i.e., there are cases where types satisfy liveness but programs do not, leading to *unsound* verification, and **(B-2)** it cannot verify infinitely spawning goroutines because either the theory is limited to bounded approximation [36] or a decidable set of types are limited to finite-control (i.e. no parallel processes inside loops) [37, 14].

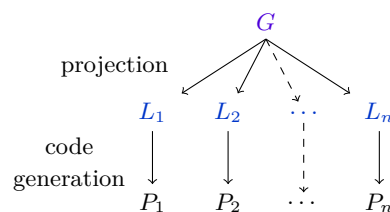
(C) Go Code Generation. Another approach is the generation of Go code from *parameterised* multiparty session protocols [3]. However, the major limitation of [3] is that participants in a protocol still need to be **fixed** at the start of a session, so it cannot express and generate code for typical Go-style programs with goroutines – e.g. a safe version of Figure 1. There is a subtle, but important distinction between dynamic participants and parameterised roles: parameterised roles cannot depend on a run-time value that is exchanged in a message *that is part of the protocol*, because in parameterised MPST approaches, all of the participants must join the session at session initialisation, and are therefore fixed.

Our challenges are to overcome all these limitations with a scalable (implementable) MPST theory. In summary, this work solves bottlenecks of the existing MPST work by proposing a new theory, **DMst**, that allows the dynamic generation of an unbounded number of participants in recursive protocols, overcoming expressiveness issues in [9, 7] **(A)** and **(C)**; unsoundness **(B-1)**, but is not limited to a bounded analysis nor finite-control **(B-2)**.

Outline. § 2 presents an overview of the GoScr toolchain; § 3 presents **DMst**, multiparty session types extended with the ability to add unbounded participants dynamically during a protocol execution, and proves its deadlock-freedom (Theorem 23), orphan message freedom, and liveness (Theorem 29); § 4 describes the code generation process of GoScr, and how to use it to implement **DMst** protocols; § 5 first measures the runtime overhead of the GoScr backend, then demonstrate the expressiveness of **DMst**, comparing the expressiveness of GoScr to **(A)** [47, 36, 37, 14] and **(B)** [3] with a number of case studies. We also implement three use cases – dynamic task delegation, a recursive Domain Name System, a noughts and crosses game with Min-Max strategy – to demonstrate the applicability of GoScr; § 6 gives related work, and § 7 concludes with future work.

2 Overview of GoScr

GoScr follows a typical Multiparty Session Types workflow (see diagram on the right). In this workflow, the starting point is the definition of a *global protocol* (*global type* in MPST), which describes a structured sequence of interactions between a number of participants. From this global type, we extract automatically a number of *local protocols* (*local types*) that describe the interactions (i.e. send or receive actions) from the point of view of every participant in the protocol. This is done using the *projection* operation. If some participant is not projectable, then we raise an error, since the protocol is not well-formed and can lead to deadlocks or other communication errors. If the programmer provides a set of processes that behaves as prescribed by each of the local types, then the whole system is safe. We take a *code generation* approach, where we generate process code from their respective local types, providing safety guarantees *by construction*.



Adding Participants Dynamically. GoScr is a code generation tool which extends nuScr [48] with the theory of DMst, targeting the Go language. nuScr is a new implementation of Scribble [51], aimed at experimenting with extensions to core MPST. Figure 2 presents an overview of GoScr. We distinguish toolchain internals (dashed boxes) from tool inputs (solid boxes). Development starts by specifying a global protocol in GoScr [51, 23], a programmer-friendly protocol description language based on MPST [22, 44]. GoScr validates the well-formedness of the protocol, and produces a local type for each participant via *projection*. GoScr generates protocol implementations from these sets of local types. We provide an overview of the GoScr workflow using the dynamic recursive pipeline of Figure 3. Intuitively, this pipeline introduces a new participant after each iteration.

Global Protocol Specification. The first key novel feature of GoScr is the ability to *define* and *call* protocols (e.g. Line 1 in Figure 3) that may bring new participants to the protocol *dynamically*, specified by the `new` keyword in the signature. These calls can be recursive, allowing for an *unbounded* number of participants. Lines 1–14 declare `UPipe`, which requires only participant `m`, and introduces a new participant `w` dynamically. Protocol calls create any necessary new participants, as well as any necessary channels, before performing the interactions described by the called protocol. The second key novel feature of GoScr is the ability to *modify* a recursive protocol by *combining* (i.e. *interleaving*) its interactions with those of a protocol call. In our syntax, this is specified by *annotating* recursion variables with protocol calls. We call this **updatable recursion**, and an example of this can be found in Line 5. The meaning of such calls is as follows. Suppose that processes r_0 and r_1 are behaving as `m` and `w` (resp.) in `UPipe`. Just before the protocol jumps back to Line 2, process r_1 calls protocol `UPipe(r1)`. This means that r_1 will create a new participant r_2 , and r_1 will *delegate* to r_2 a session to act as `w` in `UPipe`, with r_1 acting as `m`. But at this point, r_1 should act as both `m` and `w`. To address this, r_1 will combine its interactions acting as `m` and `w`. The fact that r_1 needs to change its behaviour to act as *two* distinct roles in `UPipe` will be reflected in its **local protocol specification** (participant `w` in Figure 3).

6:6 Dynamically Updatable Multiparty Session Protocols

```

1 global protocol UPipe(role M;new role W){
2   rec X {
3     choice at M {
4       (Put:int) from M to W;
5       continue X with W calls UPipe(W);
6     } or {
7       (Quit:int) from M to W; }}}
8 local protocol M@UPipe(role M;new role W){
9   rec X {
10    choice at M {
11      (Put:int) to W;
12      continue X;
13    } or {
14      (Quit:int) to W; }}}

local protocol W@UPipe(role M;new role W){
1  choice at M {
2    (Put:int) from M;
3    invite UPipe(self; new W2);
4  }
5  rec X {
6    choice at M {
7      (Put:int) from M;
8      (Put:int) to W2;
9      continue X;
10   } or {
11     (Quit:int) from M;
12     (Quit:int) to W2;
13   }}
14 } or { (Quit:int) from M; }}

```

■ **Figure 3** Global and Local protocols for a dynamic recursive pipeline.

```

1 type Put int
2 type Quit int
3 type Ctx_UPipe_W interface {
4   Recv_M_Put(v_2 Put)
5   Init_W_UPipe() Ctx_UPipe_W
6   ...
7   Recv_M_Quit(v_2 Quit)
8   Quit() }
9 func UPipeW(ctx Ctx_UPipe_W,
10 wg *sync.WaitGroup, chMW chan MsgUPipe){
11   defer wg.Done()
12   x_1 := <- chMW
13
14   switch v_2 := x_1.(type) {
15     case Put:
16       ctx.Recv_M_Put(v_2)
17       ch_W_W_1 := make(chan MsgUPipe, 1)
18       ctx_1 := ctx.Init_W_UPipe_Ctx()
19       wg.Add(1)
20       go UPipeW(ctx_1, wg, ch_W_W_1)
21     case Quit:
22       ctx.Recv_M_Quit(v_2)
23       ctx.End() }
24   }
25 }
26

```

■ **Figure 4** Implementation and context of role *W* in *UPipe* in Figure 3.

Local Protocol Specification. GoScr extracts local protocol specifications from global protocols using an operation called *projection*. Local protocols describe the structured sequence of interactions, from the point of view of a single participant. Figure 3 lists local protocols for *M* and *w*. Consider the point of view of the new participant *w* in protocol *UPipe* from Figure 3. *w* first receives an integer, either with label *Put* or *Quit* from *M*. If *w* receives *Quit*, then the protocol finishes. Otherwise, *w* performs a protocol call, bringing in a new participant *w2* to act as *w* in *UPipe*. In the subsequent interactions, from Line 5, *w* acts as both *M* (with respect to *w2*) and *w* (with respect to *M*). These lines (5 – 13) appear as a result of projecting Line 5 onto *w*. Notice that, if we have two participants, one acting as *M* and another one acting as *w*, this will generate a pipeline with an unbounded number of stages, until the first participant acting as *M* sends *Quit*. These kinds of protocols could not be represented in previous MPST theories and frameworks.

Program Logic. From local protocol specifications, GoScr generates the implementation of each role as a self-contained function. GoScr interleaves communication actions and the program logic. Communication actions in Go are a direct translation of those in local protocols: a send is a regular Go send, a receive is a regular Go receive, a choice is a type switch on a label, etc. Programmer inputs at this stage are, therefore, protocol specifications and program logic. We follow a *callback* approach similar to [42, 62] that guarantees correctness of communication *by construction*, unlike other approaches that required runtime *linearity* checks [3]. We discuss this approach in detail in §4. Figure 4 presents the code that

GoScr generates for w in UPipe . The generated implementation requires that the programmer implements the **context** interface Ctx_UPipe_W (Lines 3 – 8, Figure 4). This interface defines all the necessary callbacks to implement the program logic. Programmers can use any type definition to store a local state for each participant in the protocol, e.g.

```

1 type CtxW int // This type implements Ctx_UPipe_W, and stores the accumulated sum
2 func (c *CtxW) Recv_M_Put(v upipe.Put) { // upipe.Put is also an 'int'
3     *c += CtxW(v) }
4 ...

```

By using ctxW for implementing Ctx_UPipe_W , workers will store the sum of all the numbers that they receive, and forward their accumulated sum to the next participant. The generated code for w will signal when it has terminated (Line 11), and starts by receiving from m (Line 12). Depending on whether w receives Put or Quit , w continues with the corresponding branch (Line 14). If m sends Put , then w creates a new participant that also acts as w (with respect to the previous w). To create this participant, first a channel is created (Line 17), then a new context is created (Line 18), the participant count is increased to guarantee that execution does not end before all participants have ended (Line 19), and finally a new goroutine is created (Line 20). Otherwise, if m sends Quit , the callback for ending is called, a last callback to perform any necessary cleanup is called, and the participant ends (Lines 25 and 26).

As we show in Figure 2, from a global protocol specification GoScr produces an implementation of all of its participants. To run this generated implementation, programmers must provide the necessary types to represent protocol contexts and their required callbacks. Our code generation scheme statically ensures that implementations never lead to the errors described in § 1, i.e. there will be no **deadlocks** and **orphan messages**.

3 Dynamically Updatable Unbounded Multiparty Session Protocols

This section introduces the theory of *Dynamically Updatable Multiparty Session Types* (DMst) with examples, and proves that DMst satisfies deadlock-freedom and liveness. DMst is the formalism that underlies GoScr. To illustrate our theory, consider the dynamic pipeline of Figure 3. In this protocol, new participants are introduced into the protocol after each iteration. In DMst, we write this dynamic pipeline as follows:

$$\text{Pipe} = \lambda\langle p; \nu q \rangle. \mu t. (p \rightarrow q: \text{put}[\text{nat}]. (t \blacklozenge q \leftrightarrow \text{Pipe}(q))) + (p \rightarrow q: \text{quit}. \text{end})$$

This protocol definition requires two participants p and q . Participant q is annotated with ν to specify that it is introduced *dynamically* (a *dynamic participant*). Participant p is called a *parameter participant*. The body of the protocol specifies that it is a recursive protocol ($\mu t. \dots$), with recursion variable t , where p sends to q either put or quit . This is a choice (+), where each branch starts with $p \rightarrow q: \text{put}[\text{nat}]$ and $p \rightarrow q: \text{quit}$ respectively. If p sends put , then both participants enter a new iteration, but q *extends* the protocol by performing call to $\text{Pipe}(q \leftrightarrow \text{Pipe}(q))$ before entering the new iteration. Note that although the signature mentions two participants p and q , the call in the global type only needs to list the parameter participants. This protocol call effectively brings in a new participant to the protocol (e.g. r), creates and distributes the necessary additional channels, and extends the interactions of the protocol with those of $\text{Pipe}(q; r)$.

Introducing new interactions into an existing protocol requires to *interleave* them with the actions of this existing protocol. For example, the interactions of $\text{Pipe}(q; r)$ need to be interleaved with the remaining interactions of $\text{Pipe}(p; q)$. Our protocol specification allows two forms of interleavings: (a) sequencing all the interactions of a protocol call with the remaining interactions; and (b) alternating the actions of each iteration of two recursive protocols. We introduce a protocol construct \blacklozenge to specify the latter.

3.1 Global Types of DMst

The syntax of DMst global types (given in Definition 1) is an extension of the simplest version of MPST [60]. The novel added features are **highlighted**.

► **Definition 1** (DMst Global Types).

$$\gamma ::= \mathbf{p} \rightarrow \mathbf{q}:m[U] \mid \mathbf{p} \hookrightarrow x(\vec{\mathbf{q}}) \qquad G ::= \mathbf{end} \mid \gamma.G \mid \sum_{i \in I} G_i \mid \mu \mathbf{t}.G \mid \mathbf{t} \mid G \blacklozenge \vec{\gamma}$$

Prefixes (γ, γ', \dots) represent individual interactions between **participants**, also called **roles**², $(\mathbf{p}, \mathbf{q}, \mathbf{r}, \dots)$. There are two prefixes: messages and protocol calls. A **message** between \mathbf{p} and \mathbf{q} with **label** m and **payload type** U (e.g. `int`, `bool`, ...) is written $\mathbf{p} \rightarrow \mathbf{q}:m[U]$, or $\mathbf{p} \rightarrow \mathbf{q}:m$ whenever the payload is not relevant, e.g. when U is unit. We write $\mathbf{p} \hookrightarrow x(\vec{\mathbf{q}})$ to denote a **call** to protocol x by \mathbf{p} , with participants $\vec{\mathbf{q}} (= \mathbf{q}_1 \dots \mathbf{q}_n)$ (see **protocol definitions** below). A protocol call prefix will introduce the new interactions described by x .

Global types (G, G', \dots) denote global protocols among participants. The syntax of global types is mostly standard: **end** is **termination** and it is often omitted. \mathbf{t} denotes a **recursive variable**. **Choice** $\sum_{i \in I} G_i$ chooses any G_i , depending on the first action of each G_i (see Definition 3). **Recursive protocol** $\mu \mathbf{t}.G$ behaves as G , binding recursive variable \mathbf{t} to $\mu \mathbf{t}.G$. **Sequencing** $\gamma.G$ denotes the execution of a prefix γ , and a continuation G . The new construct $G \blacklozenge \vec{\gamma}$ denotes an **updatable protocol**, where G is extended with the interactions and participants introduced by $\vec{\gamma}$ (if any). When G is a recursive variable \mathbf{t} ($\mathbf{t} \blacklozenge \vec{\gamma}$), we often call these *updatable recursion*, or *updatable recursion variable*. We use updatable protocols to represent recursive protocols where subsequent iterations are extended with new message exchanges and/or participants. We will show in Example 6 how to use updatable recursion to represent the dynamic recursive pipeline of Figure 3.

Choice well-formedness. Standard MPST syntax only allows choices where a participant \mathbf{p} sends to another participant \mathbf{q} a distinct label in each branch. This means that \mathbf{p} and \mathbf{q} can use the label to distinguish each branch of the choice [21, 60]. DMst's syntax is more flexible, since branches can also be distinguished by distinct protocol calls. However, we still require that a single participant either sends a distinct label, or performs a distinct protocol call as the first interaction of each branch. We say that the choices that satisfy this condition are *directed*. Checking that choices are directed is necessary for well-formedness, but it is not sufficient. Protocol well-formedness is defined in a standard way later in Definition 15. To refer to the interaction that occurs in a branch, we use the *extended labels*.

► **Definition 2** (Extended Labels). We define extended labels, $\ell ::= m \mid i @ x(\vec{\mathbf{p}}; \vec{\mathbf{q}})$, where $i @ x(\vec{\mathbf{p}}; \vec{\mathbf{q}})$ identifies a protocol call as the i -th participant of x with participants $\vec{\mathbf{p}}; \vec{\mathbf{q}}$. We use participant index instead of name, since x may give different names to $\vec{\mathbf{p}}$ and $\vec{\mathbf{q}}$.

► **Definition 3** (Directed Choices). Then, we define **dc** (directed choice):

$$\mathbf{dc}(\mathbf{p}, \{\ell_i\}_{i \in I}, \sum_{i \in I} \gamma_i.G_i) = (\forall i \in I. \mathbf{inter}(\mathbf{p}, \ell_i, \gamma_i)) \text{ with all } \ell_i \neq \ell_j \text{ for } i \neq j$$

The predicate $\mathbf{inter}(\mathbf{p}, \ell_i, \gamma_i)$ states that γ_i is an interaction initiated by \mathbf{p} with extended label ℓ_i : $\mathbf{inter}(\mathbf{p}, i @ x(\vec{\mathbf{p}}; \vec{\mathbf{q}}), \mathbf{p} \hookrightarrow x(\vec{\mathbf{q}}))$, if $i \leq \mathbf{size}(\vec{\mathbf{p}}\vec{\mathbf{q}})$, and $\mathbf{inter}(\mathbf{p}, m, \mathbf{p} \rightarrow \mathbf{q}:m[U])$.

² A participant plays a *role* in the protocol, and this role is determined by the structured sequence of interactions that are allowed by the global type.

Protocol definitions ($x = \lambda\langle\vec{q}; \nu\vec{r}\rangle.G$) associate a protocol name x with a global type G , given a sequence of parameter participants \vec{q} , and a sequence of *new* participants \vec{r} (where “ ν ” means “new” [41]) that join the protocol **dynamically** (we call these *dynamic participants*). Any participant occurring in G must be bound by \vec{q} or \vec{r} . Protocol call prefixes ($x\langle\vec{q}\rangle$) only specify the parameter participants, not the dynamic ones. To refer to the global type of a definition, we write $x\langle\vec{q}; \vec{r}\rangle$, with parameter participants \vec{q} , and dynamic participants \vec{r} .

► **Example 4** (Fibonacci). The following protocol represents the interactions of an unbounded series of participants, that together compute the Fibonacci sequence:

$$Fib = \lambda\langle s, f_1, f_2; \nu f_3 \rangle. f_1 \rightarrow f_3:F[int]. f_2 \rightarrow f_3:F[int]. f_3 \rightarrow s:NF[int]. f_3 \hookrightarrow Fib\langle s, f_2, f_3 \rangle.end$$

Fib defines a protocol that recursively creates new participants (f_3 in the global type) to compute the next element of the Fibonacci sequence after receiving the results from the previous two participants (f_1 and f_2). Participant s receives all the results. Intuitively, the implementation of f_3 starts by receiving from f_1 and f_2 , sends the new Fibonacci number to s , and then creates a new participant and continues with f_2 acting as f_1 , and f_3 as f_2 . The code generated by a similar protocol is shown later in Figure 5.

Protocol calls can also be used to represent recursive protocols that are augmented dynamically with new interactions and/or participants. To represent such protocols we use updatable recursion variables. Intuitively, subsequent iterations of a recursive protocol $\mu t.G$ that contains an updatable recursion variable $t \blacklozenge p \hookrightarrow x\langle\vec{q}\rangle$ will proceed as $\mu t.G$ combined with the global type defined by x . Global types are combined by interleaving their interactions.

► **Definition 5** (Combining Recursive Global Types). *Let $cont$ be a function that computes the set of final continuations, i.e. recursion variables or end , after executing all possible prefixes: $cont(\gamma.G) = cont(G)$, $cont(\mu t.G) = cont(G) \setminus \{t\}$, $cont(\sum_{i \in I} G_i) = \cup_{i \in I} cont(G_i)$, $cont(t) = \{t\}$, $cont(G \blacklozenge \gamma) = cont(G)$, $cont(end) = \{end\}$. We define*

$$(\mu t. \sum_{i \in I} G'_i) \blacklozenge (\mu t. \sum_{i \in I} G_i) = \mu t. \sum_{i \in I} (G'_i \blacklozenge_t G_i)$$

where $G' \blacklozenge_t G = [G/t]G'$ if $cont(G') = cont(G) = \{t\}$, $G' \blacklozenge_t G = [G/end]G'$ if $cont(G') = cont(G) = \{end\}$, and is undefined otherwise.

The composition operator takes two recursive protocols with the same branching structure, and combines each of the branches using $G' \blacklozenge_t G$. This operator simply appends the interactions of G after the interactions of G' by substituting either end or t by G . Both G and G' must finish with the same last continuation, either t or end . For example:

$$((\gamma_1. end) + (\gamma_2. t)) \blacklozenge_t ((\gamma_3. end) + (\gamma_4. t)) = (\gamma_1. \gamma_3. end) + (\gamma_2. \gamma_4. t),$$

but the following case is undefined: $((\gamma_1. end) + (\gamma_2. t)) \blacklozenge_t ((\gamma_3. end) + (\gamma_4. t'))$ (if $t \neq t'$)

► **Example 6** (Dynamic Recursive Pipeline). Consider again the dynamic pipeline of Figure 3:

$$Pipe = \lambda\langle p; \nu q \rangle. \mu t. (p \rightarrow q:put[nat]. (t \blacklozenge q \hookrightarrow Pipe\langle q \rangle)) + (p \rightarrow q:quit. end)$$

A set of processes that runs according to this specification would proceed as follows. The first iteration is the same as the first iteration of $Pipe$, but without updatable recursion. This is equivalent to the following global type:

$$G_0 = \mu t. (p \rightarrow q:put[nat]. t) + (p \rightarrow q:quit. end)$$

I.e. participant p would start by sending put or $quit$ to q , and q would receive this message. Subsequent iterations will *combine* G_0 , with the result of the protocol call $(Pipe\langle q \rangle)$. Given a fresh participant r , this is as follows:

$$\begin{aligned} G_1 &= G_0 \diamond \text{Pipe}(\mathbf{q}; \mathbf{r}) = G_0 \diamond (\mu \mathbf{t}. (\mathbf{q} \rightarrow \mathbf{r}: \text{put}[\text{nat}]. (\mathbf{t} \blacklozenge \mathbf{r} \hookrightarrow \text{Pipe}(\mathbf{r}))) + (\mathbf{q} \rightarrow \mathbf{r}: \text{quit}. \text{end})) \\ &= \mu \mathbf{t}. (\mathbf{p} \rightarrow \mathbf{q}: \text{put}[\text{nat}]. \mathbf{q} \rightarrow \mathbf{r}: \text{put}[\text{nat}]. (\mathbf{t} \blacklozenge \mathbf{r} \hookrightarrow \text{Pipe}(\mathbf{r}))) + (\mathbf{p} \rightarrow \mathbf{q}: \text{quit}. \mathbf{q} \rightarrow \mathbf{r}: \text{quit}. \text{end}) \end{aligned}$$

Note that \diamond plugs in the interactions of the first (second) branch of $\text{Pipe}(\mathbf{q}; \mathbf{r})$ after the first (resp. second) branch of G_0 . This has the effect that, after each iteration of the protocol, a new participant will join the pipeline, until the first participant sends message `quit`. Such protocols could not be represented in previous MPST extensions. See §6 for a discussion.

► **Example 7** (Dynamic Ring). DMst can also be used to model protocols, such a *dynamic ring*, in which participants join a recursive ring protocol. Such dynamic rings are at the core of some well-known protocols, such as Chord and its extensions. The protocol in DMst is as follows, omitting choices and payload types for simplicity:

$$\text{Ring} = \lambda \langle \mathbf{i}, \mathbf{p}; \nu \mathbf{q} \rangle. \mu \mathbf{t}. \mathbf{p} \rightarrow \mathbf{q}: \mathbf{N}. \mathbf{t} \blacklozenge (\mathbf{q} \rightarrow \mathbf{i}: \mathbf{N}. \mathbf{i} \hookrightarrow \text{Ring}(\mathbf{i}, \mathbf{q}))$$

The entrypoint is $\text{Ring}(\mathbf{p}, \mathbf{p}; \mathbf{q})$. Subsequent iterations would be combined with new protocol calls (e.g. $\text{Ring}(\mathbf{p}, \mathbf{q}; \mathbf{r})$), producing the following sequences of interactions:

$$G_0 = \mu \mathbf{t}. \mathbf{p} \rightarrow \mathbf{q}: \mathbf{N}. \mathbf{q} \rightarrow \mathbf{p}: \mathbf{N}. \mathbf{t} \quad G_1 = \mu \mathbf{t}. \mathbf{p} \rightarrow \mathbf{q}: \mathbf{N}. \mathbf{q} \rightarrow \mathbf{q}': \mathbf{N}. \mathbf{q}' \rightarrow \mathbf{p}: \mathbf{N}. \mathbf{t} \quad G_2 = \dots$$

3.2 Asynchronous Semantics of DMst Global Types

We guarantee the processes implementing all roles in a global type G indeed behave as G . To characterise the set of behaviours that are allowed by G , we define the semantics of global types as a Labelled State Transition System. The labels are the **observable actions**:

$$\alpha ::= \mathbf{pq}! \ell \mid \mathbf{pq}? \ell \mid \mathbf{pq} \nu i @ x(\vec{\mathbf{r}}; \vec{\mathbf{s}})$$

Observable $\mathbf{pq}! \ell$ is a **send** action from \mathbf{p} to \mathbf{q} with an *extended label* (either a label or a protocol call, see Definition 2). Action $\mathbf{pq}? \ell$ is **receive** and action $\mathbf{pq} \nu i @ x(\vec{\mathbf{r}}; \vec{\mathbf{s}})$ is **participant creation** which brings in \mathbf{q} as a new participant acting as the i -th role in the protocol specified by $x(\vec{\mathbf{r}}; \vec{\mathbf{s}})$. For simplicity, we sometimes write $\mathbf{pq} \nu \ell$ to refer to participant creation, assuming that ℓ is of the form $i @ x(\vec{\mathbf{r}}; \vec{\mathbf{s}})$, for some i , x , $\vec{\mathbf{r}}$ and $\vec{\mathbf{s}}$.

Extended Global Types. We extend the global types (Definition 1) with constructs that capture intermediate states of the execution of a protocol. Note that these intermediate states only appear as a result of applying the rules of the operational semantics, and these will not need to be written by users specifying full protocols. Since extended global types are a superset of Definition 1, we will use the same meta-variable G for both and, unless we specify otherwise, all global types from now on are considered to be extended.

$$\gamma ::= \dots \mid \mathbf{p} \rightarrow \mathbf{q}: \ell[U] \mid \mathbf{p} \rightsquigarrow \mathbf{q}: \ell[U] \mid \mathbf{p} \nu(\vec{\mathbf{r}}: [i, j] @ x(\vec{\mathbf{p}}; \vec{\mathbf{q}})) \mid \triangleright[G]$$

Sending protocol call labels (e.g. $\mathbf{p} \rightarrow \mathbf{q}: i @ x(\vec{\mathbf{p}}; \vec{\mathbf{q}})$) is a form of delegation that is used to perform protocol calls (see **Notations** below). $\mathbf{p} \rightsquigarrow \mathbf{q}: \ell[U]$ means \mathbf{p} has sent a message to \mathbf{q} , yet \mathbf{q} has not received it. $\mathbf{p} \nu(\vec{\mathbf{r}}: [i, j] @ x(\vec{\mathbf{p}}; \vec{\mathbf{q}}))$ represents that \mathbf{p} creates new participants $\vec{\mathbf{r}}$, acting as the i th to j th participants in x , and $\triangleright[G]$ is the nested protocol with global type G . Intuitively, a nested protocol prefix $\triangleright[G_0]. G_1$ is equivalent to sequencing G_0 and G_1 .

Notations. We use notations to break down protocol calls into the individual interactions. Suppose that $\vec{q} = (q_1, \dots, q_n)$ and $\vec{r} = (r_1, \dots, r_m)$. We define $\text{idx}(\mathbf{p}; \vec{q})$ as $\{i\}$, if $\mathbf{p} = q_i$ with $1 \leq i \leq n$, or the empty set $\{\}$ otherwise. We define the following shortcuts:

$$\begin{aligned} \mathbf{p} \rightarrow \vec{q}; \vec{i} @ x(\vec{q}; \vec{r}) &= \mathbf{p} \rightarrow q_1; i_1 @ x(\vec{q}; \vec{r}). \dots \mathbf{p} \rightarrow q_n; i_n @ x(\vec{q}; \vec{r}) \\ \mathbf{p} \text{ call } x(\vec{q}; \vec{r}) &= \mathbf{p} \nu(\vec{r} : [n+1, n+m] @ x(\vec{q}; \vec{r})). \mathbf{p} \rightarrow \vec{q} \setminus \{\mathbf{p}\}; ([1, n] \setminus \text{idx}(\mathbf{p}; \vec{q})) @ x(\vec{q}; \vec{r}) \end{aligned}$$

Notation $\mathbf{p} \rightarrow \vec{q}; \vec{i} @ x(\vec{q}; \vec{r})$ represents a sequence of messages from \mathbf{p} to each of the $q \in \vec{q}$ with the respective extended label. These are sometimes called **invitations** to x . Notation $\mathbf{p} \text{ call } x(\vec{q}; \vec{r})$ is a sequence of actions, where \mathbf{p} first creates \vec{r} , and then sends invitations to \vec{q} , excluding itself to avoid self-communication.

Global Type Equivalence and LTS. We define the erasure of updatable recursive variables as $|t \blacklozenge \gamma|_{t'} = t$ if $t = t'$; and $|t \blacklozenge \gamma|_{t'} = t \blacklozenge \gamma$ otherwise (other cases are homomorphic). The LTS is defined up to the equivalence: (1) $\mathbf{p} \nu(); G \equiv G$; (2) $\triangleright[\text{end}]. G \equiv G$; (3) $\mu t. G \equiv [\mu t. |G|_t / t] G$, and, assuming \vec{r} fresh, (4) $\mathbf{p} \hookrightarrow x(\vec{q}). G \equiv \mathbf{p} \text{ call } x(\vec{q}; \vec{r}). \triangleright[x(\vec{q}; \vec{r})]. G$, and (5) $G \blacklozenge (\vec{\gamma}. \mathbf{p} \hookrightarrow x(\vec{q})) \equiv \vec{\gamma}. \mathbf{p} \text{ call } x(\vec{q}; \vec{r}). (G \blacklozenge x(\vec{q}; \vec{r}))$. Rules (1) and (2) capture that finished prefixes (creating an empty list of participants, or a nested ended global type) can be skipped. Rule (3) is recursion unrolling. Similarly to Example 6, subsequent iterations of the protocol will combine the body of the recursion without updatable recursion variables, with the result of the protocol calls. By this rule, recursion will be updated by protocol calls, and after the first iteration, the protocol can continue as $\mu t. |G|_t$ (possibly combined with the result of a protocol call). Rule (4) expands the sequence of a protocol call and a global type, and rule (5) *updates* a global type by first executing the specified prefixes and then continuing with G combined with the result of the protocol call. We guarantee that new roles are globally fresh by adopting a Barendregt convention on all binders, i.e. each time we access a protocol definition x , we alpha-rename the participants bound by $\nu \vec{r}$ to avoid participant name clashes. Without it, consecutive protocol calls could incorrectly introduce repeated participant names.

► **Definition 8** (Active Participants). *The active participants of a global type (prefix), $\text{pt}(G)$ ($\text{pt}(\gamma)$), is the set of participants that can perform an action in the protocol (or prefix).*

$$\begin{aligned} \text{pt}(\mathbf{p} \rightarrow \mathbf{q}; \ell[U]) &= \{\mathbf{p}, \mathbf{q}\} & \text{pt}(\mathbf{p} \hookrightarrow x(\vec{q})) &= \{\mathbf{p}\} \cup \vec{q} & \text{pt}(\mathbf{p} \rightsquigarrow \mathbf{q}; \ell[U]) &= \{\mathbf{q}\} & \gamma. G &= \text{pt}(\gamma) \cup \text{pt}(G) \\ \text{pt}(\mathbf{p} \text{ call } x(\vec{q})) &= \{\mathbf{p}\} \cup \vec{q} & \text{pt}(\mathbf{p} \nu(\vec{r} : [i, j] @ x(\vec{p}; \vec{q}))) &= \{\mathbf{p}\} \cup \vec{r} & \text{pt}(\triangleright[G]) &= \text{pt}(G) \\ \text{pt}(\text{end}) &= \text{pt}(t) = \{\} & \text{pt}(\mu t. G) &= \text{pt}(G) & \text{pt}(\sum_{i \in I} G_i) &= \bigcup_{i \in I} \text{pt}(G_i) & \text{pt}(G \blacklozenge \gamma) &= \text{pt}(G) \cup \text{pt}(\gamma) \end{aligned}$$

► **Definition 9** (LTS for Global Types). Let the **subject** of an action denote the role that performs it: $\mathbf{p} = \text{subj}(\mathbf{p} \mathbf{q} ! \ell) = \text{subj}(\mathbf{p} \mathbf{q} ? \ell) = \text{subj}(\mathbf{p} \mathbf{q} \nu \ell)$. The LTS for G :

$$\begin{array}{c} \begin{array}{c} \text{[BR-A]} \\ \frac{\forall i \in I, G_i \xrightarrow{\alpha} G'_i}{\sum_{i \in I} G_i \xrightarrow{\alpha} \sum_{i \in I} G'_i} \end{array} \quad \begin{array}{c} \text{[BR-B]} \\ \frac{G_j \xrightarrow{\mathbf{p} \mathbf{q} ! \ell_j} G'_j \quad \text{dc}(\mathbf{p}, \{\ell_i\}_i, \sum_{i \in I} G_i)}{\sum_{i \in I} G_i \xrightarrow{\mathbf{p} \mathbf{q} ! \ell_j} G'_j} \end{array} \quad \begin{array}{c} \text{[NEST]} \\ \frac{G_1 \xrightarrow{\alpha} G_2}{\triangleright[G_1]. G \xrightarrow{\alpha} \triangleright[G_2]. G} \end{array} \\ \text{[NEW]} \quad \mathbf{p}(\mathbf{r}, \vec{r} : [i, j] @ x(\vec{q}; \vec{r}')). G \xrightarrow{\mathbf{p} \nu i @ x(\vec{q}; \vec{r}')} \mathbf{p}(\vec{r} : [i+1, j] @ x(\vec{q}; \vec{r}')). G \\ \text{[SEND]} \quad \mathbf{p} \rightarrow \mathbf{q}; \ell[U]. G \xrightarrow{\mathbf{p} \mathbf{q} ! \ell} \mathbf{p} \rightsquigarrow \mathbf{q}; \ell[U]. G \quad \text{[RECV]} \quad \mathbf{p} \rightsquigarrow \mathbf{q}; \ell[U]. G \xrightarrow{\mathbf{p} \mathbf{q} ? \ell} G \quad \text{[SEQ]} \quad \frac{G \xrightarrow{\alpha} G' \quad \text{subj}(\alpha) \notin \text{pt}(\gamma)}{\gamma. G \xrightarrow{\alpha} \gamma. G'} \end{array}$$

[BR-A] specifies that if an action can be taken in all branches of a choice, it can be taken before the choice is decided. The reason is that if an action can be taken in all branches, then it must be *independent* of the choice. [BR-B] states that if the sender of a choice does an action that selects branch j , then the choice transitions to this branch. [SEQ] states that

6:12 Dynamically Updatable Multiparty Session Protocols

an action can take place in a continuation, if the action does not involve the participants of the prefix. In the prefix transitions, [SEND] (resp. [RECV]) represents a send (resp. receive) action. [NEW] specifies that a new participant \mathbf{r} of the nested protocol is created, and [NEST] represents the execution of an action in the nested global type.

► **Example 10** (DMst Semantics). Consider the following protocol, cf. Example 6:

$$Pipe = \lambda\langle \mathbf{p}; \nu \mathbf{q} \rangle. \mu \mathbf{t}. G_0 \quad \text{with } G_0 = (\mathbf{p} \rightarrow \mathbf{q}; \text{put}[\text{nat}]. (\mathbf{t} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle \mathbf{q} \rangle)) + (\mathbf{p} \rightarrow \mathbf{q}; \text{quit}. \text{end})$$

First, assuming two initial participants (\mathbf{p} and \mathbf{q}), we unfold recursion using \equiv :

$$\mu \mathbf{t}. G_0 \equiv [\mu \mathbf{t}. |G_0|_{\mathbf{t}} / \mathbf{t}] G_0 = (\mathbf{p} \rightarrow \mathbf{q}; \text{put}[\text{nat}]. (\mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle \mathbf{q} \rangle)) + (\mathbf{p} \rightarrow \mathbf{q}; \text{quit}. \text{end})$$

There are two allowed actions: sending **put** and sending **quit**. By [BR-B] and [SEND],

$$[\mu \mathbf{t}. |G_0|_{\mathbf{t}} / \mathbf{t}] G_0 \xrightarrow{\mathbf{pq}! \text{put}} \mathbf{p} \rightsquigarrow \mathbf{q}; \text{put}[\text{nat}]. (\mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle \mathbf{q} \rangle)$$

There are now two actions accepted. First, we can use [RECV]:

$$\mathbf{p} \rightsquigarrow \mathbf{q}; \text{put}[\text{nat}]. (\mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle \mathbf{q} \rangle) \xrightarrow{\mathbf{qp}^? \text{put}} \mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle \mathbf{q} \rangle$$

To enable the second action, we use equivalences to unfold the updatable global type:

$$\begin{aligned} G_1 &= \mathbf{p} \rightsquigarrow \mathbf{q}; \text{put}[\text{nat}]. (\mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge \mathbf{q} \hookrightarrow Pipe\langle \mathbf{q} \rangle) \\ &\equiv \mathbf{p} \rightsquigarrow \mathbf{q}; \text{put}[\text{nat}]. \mathbf{q} \nu (\mathbf{r} : 2@Pipe(\mathbf{q}; \mathbf{r})). (\mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge Pipe(\mathbf{q}; \mathbf{r})) \end{aligned}$$

Note that \mathbf{p} is not in the set of active participants of the prefix, so \mathbf{p} can take a step, using repeated applications of [SEQ], in $(\mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge Pipe(\mathbf{q}; \mathbf{r}))$.

$$\begin{aligned} \mu \mathbf{t}. G_2 &= (\mu \mathbf{t}. |G_0|_{\mathbf{t}} \blacklozenge Pipe(\mathbf{q}; \mathbf{r})) \\ &= \mu \mathbf{t}. (\mathbf{p} \rightarrow \mathbf{q}; \text{put}[\text{nat}]. \mathbf{q} \rightarrow \mathbf{r}; \text{put}[\text{nat}]. (\mathbf{t} \blacklozenge \mathbf{r} \hookrightarrow Pipe\langle \mathbf{r} \rangle)) + (\mathbf{p} \rightarrow \mathbf{q}; \text{quit}. \mathbf{q} \rightarrow \mathbf{r}; \text{quit}. \text{end}) \\ &\equiv (\mathbf{p} \rightarrow \mathbf{q}; \text{put}[\text{nat}]. \mathbf{q} \rightarrow \mathbf{r}; \text{put}[\text{nat}]. (\mu \mathbf{t}. |G_2|_{\mathbf{t}} \blacklozenge \mathbf{r} \hookrightarrow Pipe\langle \mathbf{r} \rangle)) + (\mathbf{p} \rightarrow \mathbf{q}; \text{quit}. \mathbf{q} \rightarrow \mathbf{r}; \text{quit}. \text{end}) \end{aligned}$$

Suppose that G_2 proceeds by \mathbf{p} sending **quit**: $\mu \mathbf{t}. G_2 \xrightarrow{\mathbf{pq}! \text{quit}} (\mathbf{p} \rightsquigarrow \mathbf{q}; \text{quit}. \mathbf{q} \rightarrow \mathbf{r}; \text{quit}. \text{end})$. Then, G_1 transitions to the following global type:

$$\mu \mathbf{t}. G_1 \xrightarrow{\mathbf{pq}! \text{quit}} \mathbf{p} \rightsquigarrow \mathbf{q}; \text{put}[\text{nat}]. \mathbf{q} \nu (\mathbf{r} : 2@Pipe(\mathbf{q}; \mathbf{r})). \mathbf{p} \rightsquigarrow \mathbf{q}; \text{quit}. \mathbf{q} \rightarrow \mathbf{r}; \text{quit}. \text{end}$$

After [SQ-A] and [RECV], the global type transitions to:

$$G_3 = \mathbf{q} \nu (\mathbf{r} : 2@Pipe(\mathbf{q}; \mathbf{r})). \mathbf{p} \rightsquigarrow \mathbf{q}; \text{quit}. \mathbf{q} \rightarrow \mathbf{r}; \text{quit}. \text{end}$$

With [SQ-A] and [NEW], the protocol transitions as follows:

$$G_3 \xrightarrow{\mathbf{qr} \nu 2@Pipe(\mathbf{q}; \mathbf{r})} \mathbf{p} \rightsquigarrow \mathbf{q}; \text{quit}. \mathbf{q} \rightarrow \mathbf{r}; \text{quit}. \text{end}$$

At this stage, \mathbf{r} is a new active participant of the protocol. The remaining global type can run to completion via a sequence of [RECV], [SEND], and finally [RECV].

3.3 Local Types

Local types describe the interactions of a protocol from the point of view of a single participant.

► **Definition 11** (DMst Local Types). Let $M ::= l[U] \mid L$. The syntax of local types is:

$$\pi ::= \mathbf{p}!M \mid \mathbf{p}?M \mid \nu(\vec{\mathbf{p}} : \vec{L}) \mid \triangleright[L] \quad L ::= \text{end} \mid \pi. L \mid \sum_{i \in I} L_i \mid \mu \mathbf{t}. L \mid \mathbf{t} \mid L \blacklozenge \vec{\pi}$$

Local type syntax differs from that of global types in the prefixes (π instead of γ). Local type prefixes are as follows: **send** $\mathbf{p}!M$, **receive** $\mathbf{p}?M$, **new participant creation** $\nu(\mathbf{p}_1 : L_1) \cdots (\mathbf{p}_n : L_n)$, and the **nested local type** $\triangleright[L]$. We lift the definitions of directed choices, updatable recursion erasure, and the composition operator from global types to local types. **Endpoint projection** takes a global type G and a participant \mathbf{r} , and produces the local type (the local interactions) of \mathbf{r} in G .

Similarly to global types, we introduce the notations for protocol calls. Assuming $\vec{\mathbf{q}} = \mathbf{q}_1, \dots, \mathbf{q}_n$ and $\vec{\mathbf{r}} = \mathbf{r}_1, \dots, \mathbf{r}_m$, we define these notations as follows:

$$\begin{aligned} \vec{\mathbf{q}}!i@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) &= \mathbf{q}_1!i_1@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}). \dots \mathbf{q}_n!i_n@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) \\ \mathbf{p} \text{ call } x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) &= \nu(\vec{\mathbf{r}} : [n+1, n+m]@x(\vec{\mathbf{q}}; \vec{\mathbf{r}})). \vec{\mathbf{q}} \setminus \mathbf{p}!([1, n] \setminus \text{idx}(\mathbf{p}; \vec{\mathbf{q}}))@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) \end{aligned}$$

► **Definition 12** (Prefix Projection). *Global type projection is defined in terms of prefix projection. Prefix projection is a partial function that takes a global prefix, and produces a possibly empty (ε) sequence of local prefixes. We give the two main rules:*

$$\begin{aligned} \mathbf{p} \rightarrow \mathbf{q}:l[U] \upharpoonright \mathbf{r} &= \begin{cases} \mathbf{q}!l[U] & \mathbf{p} = \mathbf{r} \neq \mathbf{q} \\ \mathbf{p}?l[U] & \mathbf{p} \neq \mathbf{r} = \mathbf{q} \\ \varepsilon & \mathbf{p}, \mathbf{r}, \mathbf{q} \text{ distinct} \end{cases} & \mathbf{p} \hookrightarrow x(\vec{\mathbf{q}}) \upharpoonright \mathbf{r} &= \begin{cases} \mathbf{p} \text{ call } x(\vec{\mathbf{q}}; \vec{\mathbf{r}}). \triangleright[i@x(\vec{\mathbf{q}}; \vec{\mathbf{r}})] & \mathbf{p} = \mathbf{r} \in_i \vec{\mathbf{q}} \\ \mathbf{p} \text{ call } x(\vec{\mathbf{q}}; \vec{\mathbf{r}}) & \mathbf{p} = \mathbf{r} \notin \vec{\mathbf{q}} \\ \mathbf{p}?i@x(\vec{\mathbf{q}}; \vec{\mathbf{r}}). \triangleright[i@x(\vec{\mathbf{q}}; \vec{\mathbf{r}})] & \mathbf{p} \neq \mathbf{r} \in \vec{\mathbf{q}} \\ \varepsilon & \mathbf{p} \neq \mathbf{r} \notin \vec{\mathbf{q}} \end{cases} \end{aligned}$$

$(\vec{\mathbf{r}} \text{ fresh})$

The projection of $\mathbf{p} \rightarrow \mathbf{q}:l[U]$ onto \mathbf{r} is a *send* if \mathbf{r} is \mathbf{p} , and a *receive* if \mathbf{r} is \mathbf{q} , an empty prefix if all roles are distinct, or undefined if $\mathbf{r} = \mathbf{p} = \mathbf{q}$. The projection of $\mathbf{p} \hookrightarrow x(\vec{\mathbf{q}})$ follows a similar pattern. If \mathbf{r} is \mathbf{p} , then the projected sequence of prefixes is the one that corresponds to making the protocol call, i.e. delegating channels and creating new participants. If \mathbf{r} is the i th participant in $\vec{\mathbf{q}}$, then \mathbf{r} also takes part in the protocol, so the prefixes correspond to the reception of the channel for acting as the i th participant in x , followed by the execution of the nested local type for this i th participant in x , $i@x(\vec{\mathbf{q}}; \vec{\mathbf{r}})$. If \mathbf{r} is both the protocol caller \mathbf{p} , and also takes part in it, then the prefix sequence is the sequence of prefixes for making the protocol call, followed by the nested local type for $i@x$. Note that a participant may call a protocol, and not take part in it. When this happens, the protocol caller simply distributes the necessary channels for executing the nested protocol, and then proceeds to the continuation, without entering the nested protocol.

► **Definition 13** (Projection and Merging). Projection is defined as follows:

$$\begin{aligned} \gamma.G \upharpoonright \mathbf{r} &= \gamma \upharpoonright \mathbf{r}. G \upharpoonright \mathbf{r} & \mathbf{t} \upharpoonright \mathbf{r} &= \mathbf{t} & G \blacklozenge \vec{\gamma} \upharpoonright \mathbf{r} &= \begin{cases} G \upharpoonright \mathbf{r} & (\text{if } \mathbf{r} \notin \vec{\gamma}) \\ G \upharpoonright \mathbf{r} \blacklozenge (\vec{\gamma} \upharpoonright \mathbf{r}) & (\text{if } \mathbf{r} \in \vec{\gamma}) \end{cases} \\ \mu\mathbf{t}.G \upharpoonright \mathbf{r} &= \begin{cases} \mu\mathbf{t}.G \upharpoonright \mathbf{r} & \mathbf{r} \in \text{pt}(G) \text{ or } \\ & \text{fv}(\mu\mathbf{t}.G) \neq \emptyset \\ \text{end} & \text{otherwise} \end{cases} & \sum_{i \in I} G_i \upharpoonright \mathbf{r} &= \begin{cases} \sum_{i \in I} (G_i \upharpoonright \mathbf{r}) & \text{dc}(\mathbf{p}, \vec{\ell}, \sum_{i \in I} G_i), \mathbf{r} = \mathbf{p} \\ \prod_{i \in I} (G_i \upharpoonright \mathbf{r}) & \text{dc}(\mathbf{p}, \vec{\ell}, \sum_{i \in I} G_i), \mathbf{r} \neq \mathbf{p} \end{cases} \end{aligned}$$

Projection is a partial function from global to local types. We lift the definition of directed choices (Definition 3, dc) to local types. We define $\prod_{i \in I} L_i$ as the *merging* operator:

$$\begin{aligned} (1) \quad L \sqcap L &= L & (2) \quad \mu\mathbf{t}.L_1 \sqcap \mu\mathbf{t}.L_2 &= \mu\mathbf{t}.(L_1 \sqcap L_2) \\ (3) \quad \sum_{i \in I} \mathbf{p}?\ell_i[U_i]. L_i \sqcap \sum_{j \in J} \mathbf{p}?\ell_j[U_j]. L'_j &= \\ & \sum_{k \in I \cap J} (\mathbf{p}?\ell_k[U_k]. L_k \sqcap L'_k) + \sum_{i \in I \setminus J} (\mathbf{p}?\ell_i[U_i]. L_i) + \sum_{j \in J \setminus I} (\mathbf{p}?\ell_j[U_j]. L'_j) \end{aligned}$$

The projection rules are standard [60], except the choice. A choice is only defined if it is directed. The projection of the participant that makes the choice is a local type choice of the projection of the branches. The projection for all other participants is the *merging* of the

projection of the branches. Local types can be merged in three cases: (1) they are the same, (2) they are recursive local types whose bodies can be merged, or (3) they become aware of which branch of the choice was taken (if necessary), by receive actions with distinct labels. Case (3) implies that both local types are choices with a receive prefix as the first action, where the continuations for the branches with the same labels can be merged.

It is standard in MPST to define *well-formedness* in terms of *projectability* [21]. This means that if a global type is projectable onto all of its roles, then it is well-formed and therefore live and deadlock-free. Unfortunately, the use of \blacklozenge means that this is not possible with DMst. E.g., the following global type is projectable, but it will get stuck:

$$\begin{aligned} Proto1 &= \lambda\langle \mathbf{p}; \nu \mathbf{r} \rangle. \mu \mathbf{t}. (\mathbf{r} \rightarrow \mathbf{p}:m_1. \text{end}) + (\mathbf{r} \rightarrow \mathbf{p}:m_2. \mathbf{t}) \\ IllFormed &= \lambda\langle \mathbf{p}; \nu \mathbf{q} \rangle. \mu \mathbf{t}. (\mathbf{p} \rightarrow \mathbf{q}:m_1. \text{end}) + (\mathbf{p} \rightarrow \mathbf{q}:m_2. \mathbf{t} \blacklozenge (\mathbf{p} \hookrightarrow Proto1 \langle \mathbf{p} \rangle)) \end{aligned}$$

Specifically, \mathbf{r} in *Proto1* will not become aware of the branch taken by \mathbf{p} in *IllFormed*, so after unfolding *IllFormed* once, we will obtain the following global type:

$$\mu \mathbf{t}. (\mathbf{p} \rightarrow \mathbf{q}:m_1. \mathbf{r} \rightarrow \mathbf{p}:m_1. \text{end}) + (\mathbf{p} \rightarrow \mathbf{q}:m_2. \mathbf{r} \rightarrow \mathbf{p}:m_2. \mathbf{t})$$

But this protocol would not be projectable. To avoid such cases, we define a necessary condition for well-formedness, the *safe protocol update* condition.

► **Definition 14** (Safe Protocol Update). *Suppose that $C[\]$ and $C'[\]$ are 1-hole global type contexts. A global type $\mu \mathbf{t}. C[\mathbf{t} \blacklozenge (\vec{\gamma}. \mathbf{p} \hookrightarrow x(\vec{\mathbf{q}}))]$ contains a safe update if its 1-unfolding is some $C'[G \blacklozenge (\vec{\gamma}. \mathbf{p} \hookrightarrow x(\vec{\mathbf{q}}))]$, such that given a sequence of fresh roles $\vec{\mathbf{r}}$, $G \blacklozenge x(\vec{\mathbf{q}}; \vec{\mathbf{r}})$ is projectable.*

► **Definition 15** (Projection and Well-Formed Global Types). *A global type G is projectable if its projection $G \upharpoonright \mathbf{r}$ is defined on all roles $\mathbf{r} \in G$. A global type is well formed iff it is projectable, and contains only safe protocol updates.*

► **Definition 16** (Projections of Protocol Definitions). *Assume a definition $x = \lambda\langle \vec{\mathbf{p}}; \nu \vec{\mathbf{p}}' \rangle. G$, with participants $\vec{\mathbf{p}} = (\mathbf{p}_1, \dots, \mathbf{p}_n)$ and with participants $\vec{\mathbf{p}}' = (\mathbf{p}_{n+1}, \dots, \mathbf{p}_m)$. The projections of x are the local protocol definitions that correspond to each of the participants in the protocol:*

$$1 @ x = \lambda\langle \vec{\mathbf{p}}; \nu \vec{\mathbf{p}}' \rangle. G \upharpoonright \mathbf{p}_1 \quad \dots \quad m @ x = \lambda\langle \vec{\mathbf{p}}; \nu \vec{\mathbf{p}}' \rangle. G \upharpoonright \mathbf{p}_m$$

► **Example 17** (Directed Choices and Merging). *BFib* computes the n -th Fibonacci number:

$$\begin{aligned} BFib &= \lambda\langle \mathbf{r}, \mathbf{f}_1, \mathbf{f}_2; \nu \mathbf{f}_3 \rangle. \mathbf{f}_1 \rightarrow \mathbf{f}_3: F[\text{int}]. \\ &\quad \mathbf{f}_2 \rightarrow \mathbf{f}_3: F[\text{int}]. ((\mathbf{f}_3 \rightarrow \mathbf{r}: \text{NF}[\text{int}]. \mathbf{f}_3 \rightarrow \mathbf{f}_2: \text{quit}. \text{end}) + (\mathbf{f}_3 \hookrightarrow BFib \langle \mathbf{r}, \mathbf{f}_2, \mathbf{f}_3 \rangle. \text{end})) \end{aligned}$$

This protocol is similar to that of Example 4, but instead of calling *BFib* indefinitely, the protocol offers a choice: \mathbf{f}_3 will either reply to \mathbf{r} with its Fibonacci number, or call *BFib* recursively to compute the next number. Participant \mathbf{f}_3 selects the branch of the protocol that is taken, and \mathbf{r} offers the two branches. The choice has a single sender, and both branches can be distinguished by the labels or protocol calls, so the choice is *directed* by \mathbf{f}_3 , with extended labels $\vec{\ell} = \text{NF}, i @ BFib(\mathbf{r}, \mathbf{f}_2, \mathbf{f}_3; \mathbf{f}_4)$. In a directed choice, one participant decides the branch. But how do the remaining participants *know* which branch was taken? Consider \mathbf{f}_1 in *BFib*. Its part in both branches of the protocol is the same, *end*, so we can project \mathbf{f}_1 in the choice as *end*. This is one of the cases of Definition 13: two local types can be merged if they are the same. But \mathbf{f}_2 's behaviour is different in each branch: $\mathbf{f}_3? \text{quit}. \text{end}$ and $\mathbf{f}_3? 2 @ BFib(\mathbf{r}, \mathbf{f}_2, \mathbf{f}_3; \mathbf{f}_4). \text{end}$ respectively. However, \mathbf{f}_2 is *aware* of the branch that was taken by receiving either label *quit* or protocol call label $2 @ BFib(\mathbf{r}, \mathbf{f}_2, \mathbf{f}_3; \mathbf{f}_4)$. This is case (3), as explained after Definition 13:

$$(\mathbf{f}_3?quit. end) \sqcap (\mathbf{f}_3?2@BFib(\mathbf{r}, \mathbf{f}_2, \mathbf{f}_3; \mathbf{f}_4). end) = (\mathbf{f}_3?quit. end) + (\mathbf{f}_3?2@BFib(\mathbf{r}, \mathbf{f}_2, \mathbf{f}_3; \mathbf{f}_4). end)$$

► **Example 18** (Projecting Pipeline). Consider again Example 6. We are projecting the first and second participants of x . The result of the syntactic projection is as follows:

$$\begin{aligned} x(\mathbf{p}; \mathbf{q}) &= \mu t. (\mathbf{p} \rightarrow \mathbf{q}; put[nat]. (t \blacklozenge \mathbf{q} \hookrightarrow x(\mathbf{q}))) + (\mathbf{p} \rightarrow \mathbf{q}; quit. end) \\ 1@x(\mathbf{p}; \mathbf{q}) &= \mu t. (\mathbf{q}!put[nat]. t) + (\mathbf{q}!put[nat]. end) \\ 2@x(\mathbf{p}; \mathbf{q}) &= \mu t. (\mathbf{p}?put[nat]. (t \blacklozenge (\text{call } x(\mathbf{q}; \mathbf{r}). \mathbf{q}?q1@x(\mathbf{q}; \mathbf{r}). \triangleright[1@x(\mathbf{q}; \mathbf{r}]))) + (\mathbf{p}?quit. end) \end{aligned}$$

3.4 Semantics of DMst Local Types and Correctness

The semantics for local types is defined for **local type configurations**. A configuration is a pair of *channel* and *participant* environments, $\langle \Delta ; \Theta \rangle$. The channel environment Δ contains the shared channels used for the asynchronous communication between each pair of participants, and the participant environment Θ is a set of the local types of all participants:

$$\Delta = \mathbf{p}_i \mathbf{q}_j :: \vec{w}_1, \dots, \mathbf{p}_k \mathbf{q}_l :: \vec{w}_n \quad \mathbf{w} ::= \ell[U] \quad \Theta = \{\mathbf{p}_1 :: L_1, \dots, \mathbf{q}_m :: L_m\}$$

\mathbf{w} denotes a *payload* of a message. We consider the channel and participant environments up to commutativity and associativity, since all entries must be disjoint. Channels \mathbf{pq} are channels of messages to \mathbf{p} from \mathbf{q} . We use $\Delta(\mathbf{pq})$ as notation for retrieving channel \mathbf{pq} , and $\Delta[\mathbf{pq} :: \vec{w}]$ for updating channel \mathbf{pq} with \vec{w} . Θ does not impose the ordering between the entries (like a set). We update the entry by writing $\Theta[\mathbf{p} :: L] = \mathbf{p} :: L, (\Theta \setminus \mathbf{p})$.

The semantics of configurations is defined by the LTS of local types and given in Definition 19, and it is defined up to local type equivalences, analogous to those of global types: (1) $\mu t.L \equiv \vec{\pi}. end$ if $L = t \blacklozenge \vec{\pi}$; (2) $[\mu t. |L|_t / t]L$ if $L \neq t \blacklozenge \vec{\pi}$, (3) $L \blacklozenge (\vec{\pi}. \pi) \equiv \vec{\pi}. \pi. L$, if $\pi \neq \triangleright[L']$, and (4) $L \blacklozenge (\vec{\pi}. \pi) \equiv \vec{\pi}. (L \blacklozenge L')$, if $\pi = \triangleright[L']$. The semantics of choices requires that they are directed. At the local type, all branches start with a send/receive prefix to/from the same participant \mathbf{p} . We use the predicate $dc(\mathbf{p}, \{\ell_i\}_i, \sum_{i \in I} \pi_i. L_i)$, and define it analogously to the predicate for global types.

► **Definition 19** (LTS for Local Types). The LTS for local types is defined as follows:

$$\begin{aligned} \text{[L-CONG]} \frac{\langle \Delta ; \mathbf{p} :: L \rangle \xrightarrow{\alpha} \langle \Delta' ; \Theta' \rangle}{\langle \Delta ; \mathbf{p} :: L, \Theta \rangle \xrightarrow{\alpha} \langle \Delta' ; \Theta', \Theta \rangle} \quad \text{[L-NEST]} \frac{\langle \Delta ; \mathbf{p} :: L' \rangle \xrightarrow{\alpha} \langle \Delta' ; \mathbf{p} :: L'', \Theta \rangle}{\langle \Delta ; \mathbf{p} :: \triangleright[L']. L \rangle \xrightarrow{\alpha} \langle \Delta' ; \mathbf{p} :: \triangleright[L'']. L, \Theta \rangle} \\ \text{[L-CHOICE]} \frac{j \in I \quad \langle \Delta ; \mathbf{p} :: L_j \rangle \xrightarrow{\alpha} \langle \Delta' ; \Theta \rangle \quad dc(\mathbf{q}, \vec{\ell}, \sum_{i \in I} L_i)}{\langle \Delta ; \mathbf{p} :: \sum_{i \in I} L_i \rangle \xrightarrow{\alpha} \langle \Delta' ; \Theta \rangle} \\ \text{[L-SEND]} \langle \Delta, \mathbf{qp} :: \vec{w} ; \mathbf{p} :: \mathbf{q}! \ell[U]. L \rangle \xrightarrow{\mathbf{pq}! \ell} \langle \Delta, \mathbf{qp} :: \vec{w} \cdot \ell[U] ; \mathbf{p} :: L \rangle \\ \text{[L-RECV]} \langle \Delta, \mathbf{pq} :: \ell[U] \cdot \vec{w} ; \mathbf{p} :: \mathbf{q}? \ell[U]. L \rangle \xrightarrow{\mathbf{pq}? \ell} \langle \Delta, \mathbf{pq} :: \vec{w} ; \mathbf{p} :: L \rangle \\ \text{[L-NEW]} \langle \Delta ; \mathbf{p} :: \nu(\mathbf{q}_i : L_i) \cdots (\mathbf{q}_j : L_j) \rangle \xrightarrow{\mathbf{pq}_i \nu L_i} \langle \Delta ; \mathbf{q}_i :: L_i, \mathbf{p} :: \nu(\mathbf{q}_{i+1} : L_{i+1}) \cdots (\mathbf{q}_j : L_j) \rangle \end{aligned}$$

[L-CONG] specifies a step by a participant in the configuration. [L-RECUR] unfolds recursion, and [L-CHOICE] selects one branch of a choice by performing a step into one of the continuations. In [L-CHOICE], only one action can take place in one branch, because the labels of all branches must be distinct for the choice to be directed. [L-SEND] executes a send prefix by enqueueing the label and the payload type into the channel of the receiver, [L-RECV] executes a receive prefix by dequeuing the label and payload type from the corresponding channel, [L-NEW] creates a new participant by composing its associated local type in parallel with the remainder

6:16 Dynamically Updatable Multiparty Session Protocols

of the local type environment, [L-NEST] performs a step into a nested local type. We allow the renaming of participants introduced by [L-NEW] to avoid participant name clashes. For simplicity, we assume that Δ always contains a (possibly empty) sequence of payloads for every pair of roles. For example, if \mathbf{pq} is not in Δ , we allow to match Δ with $\Delta, \mathbf{pq} :: \epsilon$.

We prove the correctness of DMst: (1) the global type semantics coincides with behaviours of local endpoints, a well-formed global type is (2) deadlock-free and (3) live. (1) together with (2) and (3) imply that the programs generated from local types projected from well-formed global types are deadlock-free and live.

We define **the projection of G** as $\llbracket G \rrbracket = \langle [] ; \mathbf{p} :: G \upharpoonright \mathbf{p}, \dots, \mathbf{q} :: G \upharpoonright \mathbf{q} \rangle$, for all $\mathbf{p}, \dots, \mathbf{q} \in \text{pt}(G)$. A configuration is a subtype of another if it contains the same participants and their local types are related under the standard subtyping relation [60], i.e., $\langle \Delta ; \Theta \rangle \leq \langle \Delta' ; \Theta' \rangle$ implies that $\Theta(\mathbf{p}) \leq \Theta'(\mathbf{p})$ for all \mathbf{p} .

► **Theorem 20** (Trace Equivalence). *If $\langle \Delta ; \Theta \rangle \leq \llbracket G \rrbracket$, then $\Gamma \vdash G \xrightarrow{\alpha^*} G'$ if and only if there exists $\langle \Delta' ; \Theta' \rangle$ such that $\langle \Delta ; \Theta \rangle \xrightarrow{\alpha^*} \langle \Delta' ; \Theta' \rangle$ and $\langle \Delta' ; \Theta' \rangle \leq \llbracket G' \rrbracket$.*

Proof. The full proof uses the extended projection, that produces both local types and the queue contents implicit in the intermediate forms. The core part of the proof is completed by induction on the derivations for the global and local type LTS, using the fact that if $G \equiv G'$, then $G \upharpoonright \mathbf{r} \equiv G' \upharpoonright \mathbf{r}$. ◀

A configuration $\langle \Delta ; \Theta \rangle$ is *final* if for all $\mathbf{pq} \in \text{dom}(\Delta)$, $\Delta(\mathbf{pq}) = \epsilon$, and for all $\mathbf{p} \in \text{dom}(\Theta)$, $\Theta(\mathbf{pq}) = \text{end}$. The configuration is in a *deadlock* if it cannot make progress and it is not final, i.e. the protocol has not ended, and all participants are stuck.

► **Definition 21** (Deadlock). $\langle \Delta ; \Theta \rangle$ is a *deadlock configuration* if there exists a sequence of actions α^* such that $\langle \Delta ; \Theta \rangle \xrightarrow{\alpha^*} \langle \Delta' ; \Theta' \rangle$, with $\langle \Delta' ; \Theta' \rangle$ not final and for all action α , $\langle \Delta' ; \Theta' \rangle \not\xrightarrow{\alpha}$.

► **Example 22** (Deadlock Configuration). A deadlock configuration is one in which the whole system can get stuck and cannot progress. A usual example of this is a configuration where all participants need to receive, but their messages have not been sent. We show below such configuration, where after one action, it reaches a receive cycle:

$$\begin{array}{l} \langle [\quad] ; \mathbf{p} :: \mathbf{q}!l[U]. \mathbf{q}?l[U]. L_1, \mathbf{q} :: \mathbf{r}?l[U]. L_2, \mathbf{r} :: \mathbf{p}?l[U]. L_3 \rangle \xrightarrow{\mathbf{pq}!l} \\ \langle [\mathbf{qp} :: l[U]] ; \mathbf{p} :: \mathbf{q}?l[U]. L_1, \mathbf{q} :: \mathbf{r}?l[U]. L_2, \mathbf{r} :: \mathbf{p}?l[U]. L_3 \rangle \not\xrightarrow{\alpha} \end{array}$$

► **Theorem 23** (Deadlock-Freedom). *If $\langle \Delta ; \Theta \rangle \leq \llbracket G \rrbracket$, then $\langle \Delta ; \Theta \rangle$ is deadlock-free.*

Proof. We show that either G is ended, or there is a step available for G , and use trace equivalence to conclude this for $\langle \Delta ; \Theta \rangle \leq \llbracket G \rrbracket$. ◀

Theorem 23 refers exclusively to the absence of *global* deadlocks, i.e. the whole system will never get stuck. But DMst also guarantees the absence of *local* deadlocks, i.e. that no participant in the system gets stuck. An example of such partial deadlocks is the usual *receive-cycle*, where a subset of participants are waiting forever, and can never make progress. DMst guarantees that this situation cannot happen. To prove this, we first show that DMst guarantees *orphan message freedom* [10], which means that all messages are eventually consumed without a type mismatch.

► **Definition 24** (Orphan Message). $\langle \Delta ; \Theta \rangle$ has an *orphan message* if there exists $\mathbf{w} \in \Delta(\mathbf{pq})$ but there exists no transition such that consumes it, i.e. there is no transition $\langle \Delta ; \Theta \rangle \xrightarrow{\alpha^*} \langle \Delta' ; \Theta' \rangle$ with $\mathbf{pq}?|\mathbf{w}| \in \alpha^*$.

► **Example 25** (Orphan Message). Orphan messages can occur whenever a send prefix is not coupled with the corresponding receive, thus leaving a message *hanging* in the corresponding buffer. For example, the following situation contains an orphan message:

$$\begin{aligned} & \langle [\quad] ; \mathbf{p} :: \mathbf{q}!l[U]. \mathbf{q}?l[U]. \text{end}, \mathbf{q} :: \mathbf{p}!l[U]. \text{end} \rangle \xrightarrow{\mathbf{pq}!l} \\ & \langle [\mathbf{qp} :: l[U]] ; \mathbf{p} :: \mathbf{q}?l[U]. \text{end}, \mathbf{q} :: \mathbf{p}!l[U]. \text{end} \rangle \xrightarrow{\mathbf{qp}!l \cdot \mathbf{pq}!l} \langle [\mathbf{qp} :: l[U]] ; \mathbf{p} :: \text{end}, \mathbf{q} :: \text{end} \rangle \end{aligned}$$

At the end of the execution, the configuration contains a non-empty buffer: $\mathbf{qp} :: l[U]$.

Another example of orphan messages is one in which the reduction gets stuck because of receiving a message of the wrong type or label, i.e. there is a reception error.

$$\begin{aligned} & \langle [\quad] ; \mathbf{p} :: \mathbf{q}!l[\text{int}]. \text{end}, \mathbf{q} :: \mathbf{p}?l[\text{bool}]. \text{end} \rangle \xrightarrow{\mathbf{pq}!l} \\ & \langle [\mathbf{qp} :: l[\text{int}]] ; \mathbf{p} :: \text{end}, \mathbf{q} :: \mathbf{p}?l[\text{bool}]. \text{end} \rangle \not\rightarrow \end{aligned}$$

In this case, reduction cannot continue, and the message $\mathbf{qp} :: l[\text{int}]$ cannot be consumed, because \mathbf{q} is expecting payload type bool .

Proving that DMst guarantees the absence of orphan messages relies on the absence of *blocked local types*. A blocked local type is a local type that contains a nested session that cannot terminate, followed by a non-empty continuation. For example, if $L = \triangleright[\mu\mathbf{t}.\mathbf{q}_2!l'[U']. \mathbf{t}]. \mathbf{p}?l[U]$, then L is blocked, because it will enter the nested protocol (with local type $\mu\mathbf{t}.\mathbf{q}_2!l'[U']. \mathbf{t}$), but it will never be able to continue executing $\mathbf{p}?l[U]$.

► **Definition 26** (Blocked Participant). A *blocked* local type is one that contains a continuation of the form $\triangleright[L_1]. L_2$, where: (a) L_1 is blocked, or (b) $L_2 \neq \text{end}$ and end is not reachable from L_1 .

► **Definition 27** (Liveness). We say that $\langle \Delta ; \Theta \rangle$ is *live*, if no participant is stuck. A participant \mathbf{p} is stuck in a configuration whenever it cannot progress, i.e. if $\Theta(\mathbf{p}) = L$ with $L \neq \text{end}$, but there is no trace $\langle \Delta ; \Theta \rangle \xrightarrow{\alpha^*} \langle \Delta' ; \Theta' \rangle$ with $\mathbf{p} = \text{subj}(\alpha)$ and $\alpha \in \alpha^*$.

► **Example 28** (Stuck Participant). The following configuration is *not live*, because even if \mathbf{p} and \mathbf{q} can continue interacting, \mathbf{r} and \mathbf{s} are stuck in a local receive cycle:

$$\begin{aligned} & \langle [\quad] ; \mathbf{p} :: \mu\mathbf{t}.\mathbf{q}!l[U]. \mathbf{t}, \mathbf{q} :: \mu\mathbf{t}.\mathbf{p}?l[U]. \mathbf{t}, \mathbf{r} :: \mathbf{s}?l[U]. L_3, \mathbf{s} :: \mathbf{r}?l[U]. L_4 \rangle \xrightarrow{\mathbf{pq}!l \cdot \mathbf{qp}!l} \\ & \langle [\quad] ; \mathbf{p} :: \mu\mathbf{t}.\mathbf{q}!l[U]. \mathbf{t}, \mathbf{q} :: \mu\mathbf{t}.\mathbf{p}?l[U]. \mathbf{t}, \mathbf{r} :: \mathbf{s}?l[U]. L_3, \mathbf{s} :: \mathbf{r}?l[U]. L_4 \rangle \xrightarrow{\alpha^*} \dots \end{aligned}$$

No possible trace can contain $\mathbf{rs}?l$ or $\mathbf{sr}?l$. Participants \mathbf{r} and \mathbf{s} are stuck. Note that, from Definition 19, only receive prefixes can get stuck, since send prefixes will always succeed.

► **Theorem 29** (Orphan Message Freedom and Liveness).

Suppose $\langle \Delta ; \Theta \rangle \leq \llbracket G \rrbracket$, such that Θ contains no blocked participants. Then $\langle \Delta ; \Theta \rangle$ is free of orphan messages and live.

Proof. Liveness is a straightforward consequence of orphan message freedom. The prefix of a local type can have two kinds of actions: outputs (sending data or invitations), or inputs (receiving data, or accepting invitations). Every input is coupled with an output by another participant (see Definition 13). Hence outputs can always be performed, in any state. To prove that inputs can always be consumed, we use trace equivalence. We show that any pending message can be received, since a step can only happen in a continuation if its subject is not in any of the previous prefixes, and it is always possible to end nested protocols, because they cannot be blocked. ◀

► **Proposition 30.** As a consequence of Theorems 23, 29 and 20, the global types of Example 10 are live and deadlock-free.

4 GoScr Code Generation

This section describes the GoScr toolchain. GoScr is an extension of nuScr [48], which is a new implementation of Scribble in OCaml. nuScr is designed with modularity and extensibility in mind, so that extensions of the core MPST theory [60] can be easily integrated.

GoScr Global Protocols. The syntax of GoScr global protocols is given in Definition 31.

► **Definition 31** (GoScr syntax).

$$\begin{aligned}
 P &::= \text{global protocol } x(\text{role } \mathbf{p}_1, \dots, \text{role } \mathbf{p}_n; \text{new role } \mathbf{q}_1, \dots, \text{role } \mathbf{q}_m) \{P^* G\} \\
 g &::= m[U] \text{ from } \mathbf{p} \text{ to } \mathbf{q} \mid \mathbf{p} \text{ calls } x(\mathbf{p}_1, \dots, \mathbf{p}_n) \\
 G &::= \text{choice at } \mathbf{p} \{G_1\} \text{ or } \dots \text{ or } \{G_n\} \mid g; G \mid \text{rec } \mathbf{t} \{G\} \mid \text{continue } \mathbf{t} \mid \text{end} \\
 &\quad \mid \text{continue } \mathbf{t} \text{ with } \{g_1; \dots; g_n; \mathbf{p} \text{ calls } x(\mathbf{p}_1, \dots, \mathbf{p}_n)\} \mid \text{do } x(\mathbf{p}_1, \dots, \mathbf{p}_n); G
 \end{aligned}$$

A GoScr module is a sequence of one or more **global** protocols. The last protocol definition is the entry point. The constructs of GoScr were chosen to mirror those defined in Definition 1: **global protocol** are protocol definitions ($x = \lambda\langle\vec{\mathbf{p}}; \nu\vec{\mathbf{q}}\rangle.G \dots$; **global protocol** can be used for protocol declarations with no new participants; **choice at p** defines directed choices from **p** to the receiver of the first interaction in the G_i ; **do** is a protocol call to a global protocol; and the rest of the constructs correspond to those of DMst. Protocol definitions in GoScr can start by defining other nested protocols, but this is simply a syntactic convenience, since we require every role in a nested protocol to be bound by the protocol signature.

Steps for Code Generation. The steps of code generation in GoScr are: (1) lifting all nested protocol definitions to the top-level; (2) obtaining the projections of all roles in all protocol definitions; (3) preprocessing local types to deal with instances of \blacklozenge (or **continue...with...** in GoScr); and, (4) translating the local types to Go functions, where communication is implemented using Go channels, interleaved with *callbacks* that will be used to implement the program logic.

Step (1) is straightforward. Step (2) is an implementation of Definition 13. Step (3) requires applying local type equivalences to unfold any updatable recursion. Step (4) traverses the local types, and generates on demand the necessary channels and callback interfaces. The type of the Go channels is an interface that represents the allowed payload types. Then, for each labelled message exchange: (1) we add a new type declaration for the label and payload type that implements the interface of allowed messages; (2) we search for a channel for the required endpoints, creating it if necessary; (3) we create the necessary callbacks before or after the interaction. The channels can be created either synchronous, or buffered with a user-specified size. Choosing synchronous channels is safe, since the traces accepted by using synchronous semantics is a subset of those accepted by our asynchronous semantics, which implies that the same safety properties will hold.

Go Code Generation. The Go code for each role and protocol is generated in `protocol/`. Communication is implemented using regular Go send/receive statements. There is no need to explicitly send message labels, since labels are encoded as type declarations. Protocol choices are encoded as type switches, either on the value returned from a previous callback (internal choices), or on the received value (external choices). We only generate implementations for branching choices that start with an explicit interaction.

```

1 func BFib_F2(ctx Ctx_BFib_F2, wg *sync.WaitGroup, ch_F2_F3, ch_F3_F2 chan MsgBFib) {
2     x := ctx.Send_F3_BFib_Fib2() // Callback to generate payload
3     ch_F3_F2 <- x // Send payload to F3
4     x_1 := <- ch_F2_F3 // External choice by from F3
5     switch v := x_1.(type) {
6     case End: // F3 chooses to finish the protocol
7         ctx.Recv_F3_BFib_End(v) // Callback for processing label End
8         ctx.End()
9         return
10    case Call_F1_BFib: // F3 sends the channel for acting as F1 in BFib
11        ctx_1 := ctx.Init_F1_BFib_Ctx() // Initialise context for F1 in BFib
12        BFib_F1(ctx_1, wg, v) // Run code for F1 in BFib with channel [v]
13        ctx.End_F1_BFib_Ctx(ctx_1) // Close context for F1 in BFib
14        ctx.End()
15        return
16    } }

```

■ **Figure 5** Implementation of role **F2** in protocol **BFib**.

Calling a nested protocol is implemented as regular Go function calls. `rec` constructs are generated as labelled `for` loops, where the body of the recursion is used to generate the body of the `for` loop, and recursive variables are translated as `continue` to the label of the corresponding variable. It is also possible to represent recursion using protocol calls. However, protocol calls would need to create the necessary channels and send them to any participant in the protocol, thus being less efficient than using `rec` and `for`.

4.1 Linearity and CFSM Code Generation

Program logic is defined through *callbacks*, similar to [42, 62], to avoid the *linearity* problem of previous Communicating Finite State Machine approaches (e.g. [3]). In a CFSM approach, code generation from a local protocol produces a series of interfaces that encode the protocol states. Each protocol state exposes *only* the permitted actions (e.g. send/receive), and returns the next state in the protocol. Programmers must use such states to implement their program logic. The linearity problem arises from the fact that nothing prevents programmers from mistakenly using the same protocol state again. For example, suppose that st_0, st_1, \dots , are protocol states that expose different send/recv actions. A programmer might (mistakenly) save state st_1 and perform its action twice in the implementation. In the Go code snippet below, st_1 is used both in Line 2 and Line 4, violating linearity:

```

1 st1 := st0.send_Msg_to_p(x)
2 st2 := st1.recv_Lbl_from_p(&z)
3 ...
4 stn := st1.recv_Lbl_from_p(&buffer) /* linearity error at st1 */

```

If participant p does not send any other message, then this implementation will **deadlock**. If p does send another message, this might cause a run-time error. A callback-based approach solves this problem *by construction*, since channels are not exposed to programmers [42, 62].

4.2 Example of Generated Go Code

Consider the following GoScr global type:

```

1 global protocol BFib(role Res, role F1, role F2; new role F3) {
2     Fib1(v:int) from F1 to F3; Fib2(v:int) from F2 to F3;
3     choice at F3 {
4         F3 calls BFib(Res, F2, F3);
5     } or {
6         Result(fib:int) from F3 to Res; End() from F3 to F2; }}

```

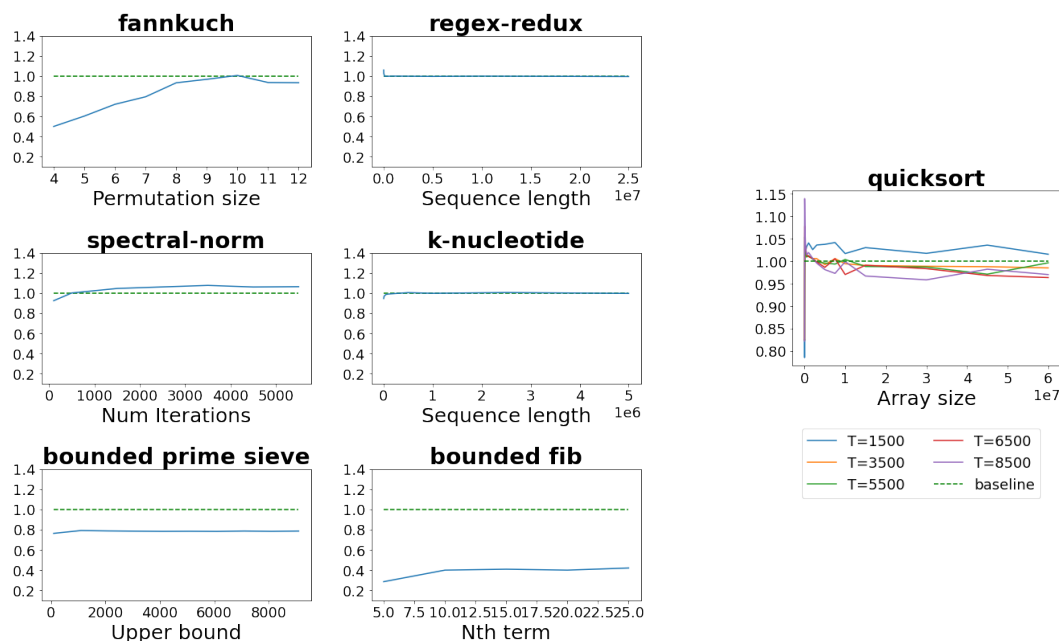
This is a *bounded* version of Example 4, that computes the Fibonacci sequence up to an upper bound. F_1 and F_2 send their respective $n - 2$ and $n - 1$ Fibonacci numbers to F_3 . Then, F_3 computes the n -th number, and makes a choice: compute the $n + 1$ number, or end the protocol. If F_3 decides to continue, then a recursive call to `BFib` happens. Otherwise, it sends the result to `Res`, and notifies F_2 that the protocol is ending. F_3 needs to notify F_2 , because depending on F_3 's decision, F_2 may need to forward its $n - 1$ number.

Figure 5 shows the code for F_2 in `BFib`. The parameters of `BFib_F2` are: `ctx` is the local state for F_2 ; `wg` is used to ensure that the main thread does not resume execution until all participants have finished executing; `ch_A_B` is the channel for communicating from B to A . The first interaction of F_2 is a message to F_3 . The payload for this message is generated in Line 2, it is sent in Line 3. Then, F_3 makes a choice: either it sends the result back to `Res` and sends `End` to F_2 to communicate the end of the protocol, or it calls `BFib` recursively. F_2 performs a type switch to check which branch it needs to take (Line 4). If the label it receives is `End` (Line 6), then F_2 processes this label and ends the protocol. Otherwise, F_2 receives an invitation as F_1 in `BFib` (Line 10); then F_2 initialises a new context for F_1 using the callback on Line 11; it calls `BFib_F1` with this new context, the waitgroup, and the received channel (Line 12); F_2 performs cleanup on the context for F_1 , gathering any necessary results from the call (Line 13); and, finally F_2 finishes.

Finally, `GoScr` generates the main protocol entrypoint, which creates the goroutines for F_1 , F_2 and F_3 , all the needed channels, and waits for the completion of the protocol.

Usability and GoScr Front-end. The tool requires the user to instantiate a large number of callbacks and interfaces to allow running a protocol. Since the `GoScr` methodology is top-down, the user must start by specifying a protocol. Therefore we expect an end-user to be aware of the callbacks and contexts that need to be instantiated. However, many of such instantiations are tedious, but straightforward, and can be automated in future work. We discuss this improvement in Section 7.

Deadlock Freedom and Liveness. Since the generated code follows the behaviour of the local types, it will satisfy both **deadlock freedom** and **liveness** (Theorems 23 and 29). Although the generated code satisfies these properties, whether the final code that is run also satisfies them depends on three requirements on the callbacks. These requirements are not checked by `GoScr`, and must be guaranteed by `GoScr` users. The three requirements that the callbacks must satisfy are: (1) callbacks must not have side-effects that interfere with other participants (e.g. using channels to add communication that is not accounted for in the protocol) (2) callbacks must be terminating, otherwise a participant may block before a necessary interaction, in a non-terminating callback; and, (3) callbacks must ensure that nested protocol calls that are not in tail position are terminating. Requirement (1) is to guarantee that programmers do not use local synchronisation mechanisms that are not accounted for in the protocol, and can cause blocking. Requirement (3) is to guarantee that any interaction after a nested protocol call is eventually performed. `GoScr` checks that local types are not blocked (Definition 26), so the code for nested calls that are not in tail position will always contain a path that ends the protocol. However, whether the actual code is terminating depends on the callback implementation that the users need to provide satisfying Requirement (3). Provided that these requirements are met, and assuming a fair scheduler, `GoScr` implementations will be deadlock free and live by construction. These requirements are not unique to our implementation. Similar requirements must be satisfied in other MPST code generation approaches.



■ **Figure 6** Execution time comparison (t_{base} / t_{GoScr}), CLBG and Quicksort.

5 Evaluation

We evaluate three aspects of GoScr: (1) the runtime overhead of the GoScr backend (§ 5.1); (2) the increased expressiveness with respect to related approaches (§ 5.3); and (3) the applicability of GoScr for building realistic protocols, by implementing dynamic task delegation, a Domain Name System, and a parallel Min-Max strategy. We show that for computation-intensive protocols, the runtime overhead of GoScr is negligible.

5.1 Runtime Overhead of GoScr

We use an Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz processor with 4 physical cores, 16GB RAM, running Ubuntu 16.04.7 and Go version go1.15.11. We use Golang’s time package to measure execution times. There are two main sources of run-time overheads: (1) callbacks; and (2) type switches and assertions. Our approach is to compare GoScr implementations against baseline Go code. Baselines are taken from benchmarking repositories, and follow similar communication patterns to the GoScr implementations. The measured time includes session initialisation. We execute each benchmark for a minimum of 20 iterations and a minimum of 20 seconds. The standard deviation for computationally expensive benchmarks is less than 5%. Only the standard deviation of fibonacci and prime sieve with small inputs (< 10th term, bound < 2000) remain high, at 70%. This is because these benchmarks with very short execution times (in the order of nanoseconds) are highly dependent on the system (e.g. channel creation, goroutine scheduling, etc). Our benchmarks are mainly taken from the *Computer Language Benchmarks Game* [17], and we include a parallel Quicksort that showcases the handling unbalanced workloads. Figure 6 shows the execution time of the Go baseline relative to GoScr: t_{base} / t_{GoScr} (below $y = 1$ is a slowdown, above is a speedup).

```

1  global protocol DynTaskGen(role S;
2     new role W) {
3     choice at S {
4         Req(req: string) from S to W;
5         S calls DynTaskGen(S);
6         choice at W {
7             Resp(resp: string) from W to S;
8         } or {
9             Error(err: string) from W to S; }
10    } or {
11        LastReq(req: string) from S to W;
12        choice at W {
13            Resp(resp: string) from W to S;
14        } or {
15            Error(err: string) from W to S;
16        }}}
11 global protocol ClientServer(role C,
12     role S) {
13     rec REPEAT {
14         Req(req: string) from C to S;
15         S calls DynTaskGen(S);
16         choice at S {
17             Resp(resp: string) from S to C;
18             continue REPEAT;
19         } or {
20             Error(err: string) from S to C;
21             continue REPEAT;
22         }}}

```

■ **Figure 7** GoScr protocol for Dynamic Task Generation.

Computer Language Benchmarks Game (CLBG). CLBG [17] is a repository of programs used to compare the performance of different languages. We use four concurrent Go programs: (1) **fannkuch** counts the maximum number of flips for a permutation of length n ; (2) **regex** matches regex patterns in a DNA string; (3) **spect** (spectral-norm) calculates the greatest eigenvalue of a matrix; and (4) **k-nuc** (k-nucleotide) counts the occurrences of a molecule sequence in a DNA string. We selected these benchmarks out of [17] because they parallelise the work using goroutines and channels, following a similar scatter/gather approach that depends on runtime values, and they could not be accurately captured by previous MPST approaches. We use the CLBG implementations [17] as the Go baseline implementations, and we extracted the communication structure of the baseline implementations as GoScr protocols. A single execution for each of these protocols takes between 1 millisecond–10 seconds depending on the input size. Smaller input sizes imply smaller local computation times, and therefore, the overhead introduced by GoScr will be more significant. We can observe a slowdown of up to 50%, in **fannkuch**, for executions in the order of magnitude of milliseconds. However, as the workload increases, the difference in the execution time shrinks to the point of becoming negligible, as we can observe in Figure 6. The **regex** baseline has a high standard deviation, which explains the small peak for the first result of **regex**, since when the execution time is in the order of hundreds of microseconds, the non-deterministic scheduling of the goroutines can significantly affect the results. **spect** seems to show that for large enough values, the GoScr implementation performs better than its naively implemented counterpart. However, the real difference in the execution time is negligible, and it is explained by differences in the program structure, e.g. the baseline uses a single shared channel, whereas GoScr generates different channels for every new goroutine.

Microbenchmarks. Bounded fibonacci (**fibonacci**) shows, as expected, that the overhead of performing type switches and callbacks is relatively high when compared with a simple addition. The baseline runs in in 40% of the execution time of GoScr. Bounded prime sieve (**prime**) shows that, when the computation complexity increases slightly (modulus operation on a stream of values), then the GoScr version performs in about 80% the execution time of the baseline. In both cases, when we add more participants and interactions to the protocol (larger values on the x-axis) the overhead *remains constant, and does not increase*.

Unbalanced Workload. In *Parallel QuickSort* (`qsort`), workers either partition the array and spawn two new workers, or apply a sequential Quicksort, depending on a threshold size (T). The execution times are similar to the CLBG benchmarks (50 microseconds–2 seconds). We observe a negligible difference in the execution time for different threshold sizes, and a spike for small arrays due to the high standard deviation for array sizes under the threshold. GoScr execution times are in the range of 1.05 and 0.95 times the baseline.

5.2 Use Cases

We demonstrate the expressiveness of GoScr using three applications, all of which require dynamic participants, and could not be expressed by previous work [3, 37].

- (a) **Dynamic Task Generation:** We present a correct implementation of the program in Figure 1 using GoScr. It is a master-worker pattern with dynamic participants.
- (b) **Domain Name System (DNS) protocol:** We demonstrate how GoScr can be used to specify one of the core Internet protocols, modelling as dynamic participants the different DNS servers which may need to be contacted in order to resolve a host’s IP address.
- (c) **Noughts and Crosses with Min-Max [49]:** We implement a Min-Max strategy for the well-known two-player game of Noughts and Crosses to demonstrate the suitability of DMst to model a parallel Divide and Conquer paradigm.

Dynamic Task Generation. The aim of this program is to generate the first n square numbers by delegating the calculation of each square number to a different worker goroutine. The program uses a common computation in Go, the *master-worker pattern*, where goroutines dynamically divide and delegate part of their tasks to other goroutines, aggregating their partial results to produce the complete result. We highlighted in § 1 (Figure 1) how even in such a simple example, incorrect management of channels can lead to orphan messages and deadlocks. Figure 7 shows a GoScr protocol specification whose behaviour is a safe version of the program in Figure 1. Notice how the behaviour of the `select` statement in Figure 1 is represented as a choice. In Figure 7, the `ClientServer` protocol models the behaviour of the main loop of the program, where two roles, a client and a server, repeatedly exchange requests (Line 14) and responses (Line 17). The server may also communicate an error in the computation of the request to the client (Line 20). We model the master-worker pattern as a call to protocol `DynTaskGen` (Line 15). Every call to the protocol introduces a new worker (`w`), and the master (`s`) will delegate a task to each new worker (Lines 4,11). If there are more tasks to assign, it will assign those tasks to new workers through recursive calls to `DynTaskGen` (Line 5). Once it has assigned the final task (Line 11), it will traverse the protocol stack, aggregating the results from the different workers in reverse order (Lines 7,13). While computing their subtask, the workers may encounter an error which they will communicate back to the server (Lines 9, 15). As opposed to the original program in Figure 1, the server will continue aggregating all the results from the workers even after encountering an error in order to ensure that there are no orphan messages.

5.3 Expressiveness

We compare the expressiveness of GoScr against the parameterised `Scribble` [3] and the static analysis framework of Go [37]. For a reference purpose, we also list comparisons with **theory-only work** in [7, 9] (i.e., they are **not** implemented). See § 6 for more detailed comparison with [7, 9]. In Table 1, we present the protocols that we implemented and whether or how closely other approaches [37, 3] can represent them. All our DMst-based implementations introduce dynamic, possibly unbounded participants. All representable

■ **Table 1** Comparison of Expressiveness.

Protocol	Dyn	Unb	Inv	DMst	[3]	[37]	[7]	[9]
1. Dynamic Ring	●		●	✓	X	X	✓	X
2. Dynamic Pipeline	●		●	✓	X	X	✓	X
3. Dynamic Recursive Pipeline	●	●		✓	X	X	X	X
4. Dynamic Recursive Tree	●	●		✓	X	X	X	X
5. Dynamic Recursive Task Gen.	●	●		✓	X	X	X	X
6. Dynamic Fork-Join	●			✓	X	X	✓	X
7. Recursive Fork-Join	●			✓	X	X	✓	X
8. Bounded Fibonacci [3]	○		○	✓	△	△	✓	X
9. Unbounded Fibonacci	●	●		✓	X	X	✓	X
10. Fannkuch-Redux [17]	○		○	✓	△	△	✓	✓
11. Spectral-Norm [17]	○			✓	△	△	✓	✓
12. Regex-Redux [17]	○			✓	✓	✓	✓	✓
13. K-Nucleotide [17]	○			✓	✓	✓	✓	✓
14. Bounded Prime Sieve	●			✓	X	X	✓	X
15. Dynamic Task Generation	●	●		✓	X	X	✓	X
16. Domain Name System [28]	●		●	✓	X	X	✓	X
17. Noughts and Crosses [42, 49]	●		●	✓	X	X	✓	X

Dyn: Dynamic participants; Unb: Unbounded participants; Inv: Choice through invitations

protocols (✓) by DMst in Table 1 are deadlock-free and live. For protocols which can be modified and re-implemented with [37] or [3], we use ○. Protocol 3 cannot be captured by any of the previous work, since it requires the dynamic introduction of participants to a recursive protocol. [3, 37] could only precisely model Protocols 12 and 13, as they create all the participants at the start. In Protocols 8, 10 and 11, the goroutines are spawned and assigned tasks dynamically, but [3, 37] can model them by initialising all goroutines at the start. We write △ to represent such changes to protocol structure. Three use cases (Protocols 13–15) discussed in § 5.2, could not be expressed by [3, 37]. In summary, DMst is more expressive than [3, 37], and capture more closely the typical Go programming style.

6 Related Work

There are a vast amount of studies of session types [27, 15, 1]. Due to the space limitations, we only compare with the most closely related work on multiparty session types (MPST).

Binary Session Types. While Scalas et al. [50] prove that the MPST processes can be mimicked by linearly typed processes with a continuation-passing style translation, in general, it is not possible to guarantee deadlock-freedom for more than two interleaved binary session processes unless one uses additional sophisticated means such as a global causal analysis on channels (e.g. [12, 4, 5]), graph-connectivity analysis with extensions on fork primitives [29], and event-driven constructs [57, 24, 34]. GV, a linear functional language with binary session types, can guarantee deadlock freedom by relying on linear typing [58]. However, linear typing prevents cyclic topologies that change dynamically, since this would require a participant to drop their communication channels when new participants join, as in Example 7. There are further substantial differences with our approach. First, GV is an end-point calculus, whereas DMst’s global types are *global specifications*, from which we can extract endpoint Go code (GoScr). Secondly, while both GV and DMst support similar programming patterns (e.g. pipeline and tree-like topologies, and channel passing), there are two major differences. Both GV and GoScr support sending effectful functions over channels (e.g. using `chan func()` type in Go, and passing a generated protocol implementation), GV’s type system would guarantee deadlock-freedom, but in Go, it would depend on how the function is used (requirements 1-3 in Section 4).

Code Generation and Multiparty Session Types. We follow the standard MPST top-down *specification-guided* methodology to guarantee **safety and liveness properties by construction using code generation**, extending an extensible toolchain, nuScr [48]. Safety by construction via code generation is a common approach in MPST. Scribble is a language/tool [51, 48] used for generating APIs for safely implementing distributed systems written in the end-point programming language that are guaranteed to conform to a protocol, and are therefore deadlock-free [25]. This approach has been applied to several languages, e.g. Scala [50, 57], Java [33], F# [43], Go [3], TypeScript [42], F \star [62] and Rust [35, 6]. A later extension of [25] proposed *explicit connection actions* as part of the Scribble protocol [26], which is also recently applied to domain-specific language in [19]. This construction specifies the point in the protocol where the different participants join, but the role of these participants must be statically known. Hence it does not allow the unbounded participants to change the protocol topology, as DMst does. *Parameterised multiparty session types* extend MPST with a parametric number of participants [11]. One example is the work by Castro-Perez et al. [3], discussed it in § 1. *Pabble* [46, 45] is another parameterised extension of Scribble used for generating safe by construction C+MPI code. Zhou et al. [62] formalised and implemented an extension of MPST with *refinement types*, which can specify constraints in the messages. Their backend targets F \star , and follows a similar callback approach to the one in this paper. Miu et al. [42] define an extension of MPST for web programming in TypeScript that uses the callback approach. Unlike DMst, the participants in all these approaches are fixed from the start of the protocol. Viering et al. [57] present a theory and implementation of MPST aimed at programming correct fault-tolerant distributed systems that supports the dynamic replacement of participants in a protocol. In their work, the replacement of participants must happen within some known roles, and their global types do not allow to extend the current protocol interactions with those of new participants. Viering et al. [57] use event handlers in their code generation, which allows safe session interleaving. Instead, we use an operator to *combine* global types in a way that does not introduce deadlocks. All previous work, unlike DMst, does not support dynamically growing protocols with an unbounded number of participants such as Example 6. Jacobs et al. [30] extend GV, a binary session typed calculus with multiparty session types. The calculus allows the introduction of new participants, but the protocols themselves are restricted to a fixed set of participants. Their use of linearity prevents the definition of recursive dynamic topologies, unlike DMst.

Dynamic Multiparty Session Types. Dynamic multirole session types (MRST) enable a set of participants which belong to the same group (i.e. role) to join a multiparty session type [9]. The major limitations are: (a) all the roles are fixed at the start (b) participants can only join at specific points in the protocol: (1) at the beginning of each iteration of a recursive protocol; or (2) at particular points marked with explicit barriers and locks. We list a number of protocols that cannot be represented using MRST in Table 1. In contrast, DMst allows any arbitrary role to join at any nested session call. A nested session call is a form of delegation, which is not supported by MRST. Therefore, a protocol such as a dynamically growing pipeline (e.g. Fibonacci in Example 4) cannot be represented by [9] either, since it would require participants to evolve their behaviour through channel passing. Nested multiparty session types [7] allow multiparty protocols with unbounded, dynamic participants. However, [7] cannot represent *recursive* protocols that are *updated* with new dynamic participants. Hence the main example of this paper, Example 6, is not representable in [7]. Moreover, nested multiparty session types cannot prove liveness (our Theorem 29), except for non-recursive protocols. Arbitrary session interleaving in [7] can introduce orphan

messages. DMst has proven deadlock-freedom and liveness clearly identifying the conditions (Definition 26). This limitation is stated in [7, Proposition 3], i.e. a protocol that violates liveness will be accepted in [7], but not in DMst. Additionally, the theory of DMst has a number of differences that make it better suited for implementing than nested MPST: (a) DMst’s choices are more flexible than those in nested MPST, since DMst can also depend on protocol calls; (b) the semantics of nested MPST is synchronous, while DMst is asynchronous; (c) nested MPST does not prove trace equivalence between global types and local type configurations; (d) The syntax of DMst’s global types are simpler than those in nested MPST, but more expressive – this is because in nested MPST, protocol definitions are part of the global type syntax, which requires the use of a kinding relation for checking well-formedness. Nested MPST protocols do not allow the occurrence of free roles, and are therefore equivalent to DMst’s global types with just top-level protocol definitions, which avoids the kinding relation for checking well-formedness. Due to our simpler but more expressive treatment, DMst is more suitable for real language implementations.

Verification of Go Programs. Our work aims at providing *correctness by construction*. The comparison with the previous code generation approach in Go [3] can be found in (C) in § 1 and **Expressiveness** in § 5.3. All of the previous work is limited to *bounded* participants. The following are several recent lines of work on *a posteriori* verification of message passing in existing Go programs. All of them use *whole-program* techniques, and support only the built-in Go channel *primitives* (i.e., intra-process messaging); none of them, however, support a dynamic, unbounded number of participants. Gobra is an automated tool for the modular verification of Go programs, based on separation logic [59]. Gobra is aimed at the functional verification of Go programs, whereas our approach focuses on communication safety. GoScr is fully automated, and aimed at building live and deadlock-free communicating systems by construction. In contrast, Gobra is aimed at the verification of annotated Go code, and it requires a high amount of invariant annotations.

Ng and Yoshida [47] extract graph-based protocol specifications [38] from Go programs that are checked for deadlock-freedom; Stadtmüller et al. [53] extract regex-based protocol specifications [55], checked for deadlock-freedom. Both approaches work only for programs restricted to *synchronous* Go channels; the former also requires all goroutines to be spawned before any communication among them occurs, and the latter has limited support for branching behaviours. Lange et al. [36, 37] (already compared in (B) in § 1 and **Expressiveness** in § 5.3) statically infer channel communication patterns from Go programs as *behavioural types*, that are checked for liveness properties. This was recently extended to analyse shared memory concurrency [14]. Like previous work, their tool is also limited to verify finite controlled programs, it is best-effort only due to the imprecision of the inference, and the verification times (and timeouts) preclude practical checking on the fly during programming. Liu et al. [39] present a tool that detects blocking misuse-of-channel bugs in Go and produces bug fixes for Go programs. Unlike DMst, Liu et al. [39] focuses only on practice, and does not formalise nor guarantee communication safety, deadlock-freedom nor liveness. Moreover their tool produces both false positive and false negative errors.

7 Conclusion and Future Work

GoScr is the first implementation of multiparty session types with dynamic, unbounded participants, from which we generate Go code with unbounded participants that is, *by construction*, deadlock-free and live. GoScr focuses on correctness (Theorems 20, 23 and

29), and it is strictly more expressive than previous Go verification frameworks (see § 1, Table 1, § 6). Furthermore, we observe that whenever the computation time is large with respect to the communication time, the performance overhead becomes negligible. GoScr is therefore suitable for implementing systems where correctness is prioritised, or systems where the computation times dominate over communication. Currently, DMst does not allow a participant to communicate with an unbounded number of participants during protocol execution. This is a limitation of the Go code generation, which we plan to address in future work. We are also considering extending our back-end to use event-handlers in the style of Viering et al. [57], and allow the arbitrary parallel composition of global types instead of our combination operator. We are also planning to extend the back-end to disparate transports (e.g. using TCP instead of Go channels), thus allowing the implementation of distributed systems. The main challenge of this is integrating delegation, as it is required by protocol invitations, in these disparate transports. Finally, to simplify usability, we plan to extend the protocol specification with annotations to guide code generation, so we can automatically generate trivial callback/context instantiation. We plan to draw inspiration for such annotations from *choreographies*, e.g [31].

References


- 1 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/25000000031.
- 2 Lorenzo Bettini, Mario Coppo, Loris D’Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR 2008 - Concurrency Theory*, pages 418–433. Springer, 2008.
- 3 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go. In *POPL’19*. ACM, 2019. doi:10.1145/3290342.
- 4 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *15th International Conference on Coordination Models and Languages*, volume 7890 of *LNCS*, pages 45–59. Springer, 2013.
- 5 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS*, 26:238–302, 2015.
- 6 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693. ACM, 2022.
- 7 Romain Demangeon and Kohei Honda. Nested protocols in session types. In *CONCUR 2012 - Concurrency Theory*, pages 272–286. Springer, 2012.
- 8 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods in System Design*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.
- 9 Pierre-Malo Deniérou and Nobuko Yoshida. Dynamic multirole session types. In *POPL’11*, pages 435–446. ACM, 2011.
- 10 Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP 2013*, volume 7966 of *LNCS*, pages 174–186. Springer, 2013. doi:10.1007/978-3-642-39212-2_18.

- 11 Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.
- 12 Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Nobuko Yoshida. On progress for structured communications. In *TGC 2007*, volume 4912 of *LNCS*, pages 257–275. Springer, 2007. doi:10.1007/978-3-540-78663-4_18.
- 13 Docker: Empowering app development for developers. <https://www.docker.com/>, November 2020.
- 14 Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs. In *ECOOP'20*, *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
- 15 Simon Gay and Antonio Ravara, editors. *Behavioural Types: from Theory to Tools*. River Publishers series in automation, control and robotics. River Publishers, June 2017.
- 16 Github 2.0: A small place to discover languages in github. https://madnight.github.io/github/#/pull_requests/2020/3, 2020.
- 17 Issac Gouy. Computer language benchmark game. <http://benchmarksgame.alioth.debian.org>, 2017.
- 18 gRPC - a high-performance, open source universal rpc framework. <https://grpc.io/>, November 2020.
- 19 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In *ECOOP 2021*, volume 194 of *LIPICs*, pages 10:1–10:30. Schloss Dagstuhl, 2021. doi:10.4230/LIPICs.ECOOP.2021.10.
- 20 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., 1985.
- 21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proc. of 35th Symp. on Princ. of Prog. Lang.*, POPL '08, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.
- 22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.
- 23 Raymond Hu. Distributed programming using Java APIs generated from Session Types. *Behavioural Types: from Theory to Tools*, pages 287–308, 2017.
- 24 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-Safe Eventful Sessions in Java. In *ECOOP 2010*, volume 6183 of *LNCS*, pages 329–353. Springer, 2010.
- 25 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE 2016*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016. doi:10.1007/978-3-662-49665-7_24.
- 26 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE 2017*, volume 10202 of *LNCS*, pages 116–133. Springer, 2017. doi:10.1007/978-3-662-54494-5_7.
- 27 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostros, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.
- 28 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In *ECOOP 2020*, volume 166 of *LIPICs*, pages 9:1–9:30. Schloss Dagstuhl, 2020. doi:10.4230/LIPICs.ECOOP.2020.9.
- 29 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. doi:10.1145/3498662.
- 30 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022. doi:10.1145/3547638.

- 31 Sung-Shik Jongmans and Petra van den Bos. A Predicate Transformer for Choreographies - Computing Preconditions in Choreographic Programming. In *ESOP 2022*, volume 13240 of *LNCS*, pages 520–547. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.
- 32 Kubernetes: Production-grade container orchestration. <https://kubernetes.io/>, June 2017.
- 33 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with Mungo and StMungo. In *PPDP*, pages 146–159, 2016. doi:10.1145/2967973.2968595.
- 34 Dimitrios Kouzapas, Nobuko Yoshida, Raymond Hu, and Kohei Honda. On Asynchronous Eventful Session Semantics. *MSCS*, 29:1–62, 2015.
- 35 Nicolas Lagailardie, Rumyana Neykova, and Nobuko Yoshida. Implementing Multiparty Session Types in Rust. In *Coordination Models and Languages*, volume 12134, pages 127–136. Springer, 2020. doi:10.1007/978-3-030-50029-08.
- 36 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. In Giuseppe Castagna and Andrew D. Gordon, editors, *POPL 2017*, pages 748–761. ACM, 2017.
- 37 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *40th International Conference on Software Engineering*, pages 1137–1148. ACM, 2018.
- 38 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232. ACM, 2015. doi:10.1145/2676726.2676964.
- 39 Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in Go software systems. In *ASPLOS '21*, April 2021.
- 40 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.
- 41 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, 1992. doi:10.1016/0890-5401(92)90009-5.
- 42 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in TypeScript with Routed Multiparty Session Types. In *CC 2021*, 2021.
- 43 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: Compile-time API generation of distributed protocols with refinements in F#. In *CC 2018*, pages 128–138. ACM, 2018. doi:10.1145/3178372.3179495.
- 44 Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. *Models, Languages, and Tools for Concurrent and Distributed Programming*, 11665:236–259, 2019. doi:10.1007/978-3-030-21485-214.
- 45 Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. Protocols by default - safe MPI code generation based on session types. In *CC 2015*, volume 9031 of *LNCS*, pages 212–232. Springer, 2015. doi:10.1007/978-3-662-46663-6_11.
- 46 Nicholas Ng and Nobuko Yoshida. Pabble: parameterised scribble. *Service Oriented Computing and Applications*, 9(3-4):269–284, 2015. doi:10.1007/s11761-014-0172-8.
- 47 Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *CC 2016*, pages 174–184. ACM, 2016. doi:10.1145/2892208.2892232.
- 48 The nuScr authors. nuscr homepage. <https://nuscr.github.io/nuscr/>, 2019.
- 49 Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Pearson Education, 2016.
- 50 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP 2017*, volume 74 of *LIPICs*, pages 24:1–24:31. Schloss Dagstuhl, 2017. doi:10.4230/LIPICs.ECOOP.2017.24.
- 51 Scribble Authors. Scribble: Describing multiparty protocols. <http://www.scribble.org/>, 2015. Accessed in Nov. 2020.
- 52 Stack overflow developer survey 2020. <https://insights.stackoverflow.com/survey/2020>, 2020.

- 53 Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. Static trace-based deadlock analysis for synchronous mini-go. In *APLAS 2016*, volume 10017 of *LNCS*, pages 116–136, 2016. doi:10.1007/978-3-319-47958-3_7.
- 54 I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003. doi:10.1109/TNET.2002.808407.
- 55 Martin Sulzmann and Peter Thiemann. Forkable regular expressions. In *LATA 2016*, volume 9618 of *LNCS*, pages 194–206. Springer, 2016. doi:10.1007/978-3-319-30000-9_15.
- 56 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiyang Zhang. Understanding real-world concurrency bugs in go. In *ASPLOS 2019*, pages 865–878. ACM, 2019. doi:10.1145/3297858.3304069.
- 57 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *PACMPL*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485501.
- 58 Philip Wadler. Propositions as Sessions. In *ICFP'12*, pages 273–286. ACM, 2012.
- 59 Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. Gobra: Modular specification and verification of Go programs. In *CAV*, pages 367–379. Springer, 2021.
- 60 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to Multiparty Session Types. In *16th International Conference on Distributed Computing and Internet Technology*, volume 11969 of *LNCS*, pages 73–93. Springer, 2020.
- 61 T. Yuan, G. Li, J. Lu, C. Liu, L. Li, and J. Xue. GoBench: A benchmark suite of real-world Go concurrency bugs. In *CGO 2021*. ACM/IEEE, 2021.
- 62 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Romyana Neykova, and Nobuko Yoshida. Statically Verified Refinements for Multiparty Protocols. *PACMPL*, 4(OOPSLA), 2020.


Modular Compilation for Higher-Order Functional Choreographies

Luís Cruz-Filipe ✉ 

Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Eva Graversen ✉ 

Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Lovro Lugović ✉ 

Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Fabrizio Montesi ✉ 

Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Marco Peressotti ✉ 

Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Abstract

Choreographic programming is a paradigm for concurrent and distributed software, whereby descriptions of the intended communications (choreographies) are automatically compiled into distributed code with strong safety and liveness properties (e.g., deadlock-freedom).

Recent efforts tried to combine the theories of choreographic programming and higher-order functional programming, in order to integrate the benefits of the former with the modularity of the latter. However, they do not offer a satisfactory theory of compilation compared to the literature, because of important syntactic and semantic shortcomings: compilation is not modular (editing a part might require recompiling everything) and the generated code can perform unexpected global synchronisations.

In this paper, we find that these shortcomings are not mere coincidences. Rather, they stem from genuine new challenges posed by the integration of choreographies and functions: knowing which participants are involved in a choreography becomes nontrivial, and divergence in applications requires rethinking how to prove the semantic correctness of compilation.

We present a novel theory of compilation for functional choreographies that overcomes these challenges, based on types and a careful design of the semantics of choreographies and distributed code. The result: a modular notion of compilation, which produces code that is deadlock-free and correct (it operationally corresponds to its source choreography).

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Distributed computing models; Computing methodologies → Distributed programming languages

Keywords and phrases Choreographies, Concurrency, λ -calculus, Type Systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.7

Related Version *Full Version:* <https://arxiv.org/abs/2111.03701>

Funding This work was partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.



© Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti;
licensed under Creative Commons License CC-BY 4.0

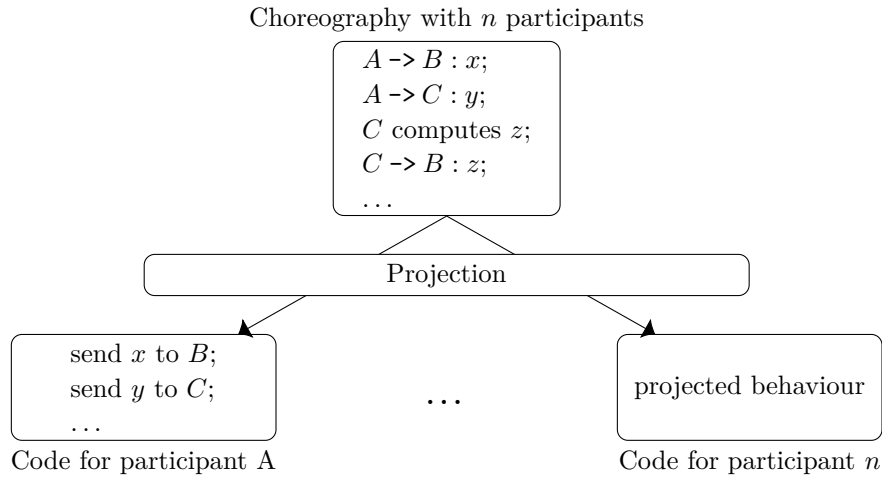
37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 7; pp. 7:1–7:37

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Choreographic programming: the communication and computation behaviour of a system is defined in a choreography, which is then projected (compiled) to deadlock-free distributed code (adapted from [17]).

1 Introduction

Functional and choreographic programming

Higher-order functional programming is a popular paradigm, which allows programmers to write modular code with strong guarantees through types. However, when dealing with concurrent and distributed programs, functional programming still requires developers to manually write a separate program for each participant, using send and receive actions to communicate data. This makes it easy to write programs that deadlock, or perform in other unexpected ways [22].

Choreographic programming (Figure 1) is a simple and powerful method to produce distributed code that does what it is supposed to do [23, 21, 18]. In this paradigm, programs are choreographies: structured compositions of the intended communications and computations that participants should perform, given from a joint perspective. A communication is expressed in some variation of the communication term from security protocol notation, Alice \rightarrow Bob : M , which reads “Alice communicates the message M to Bob” [26]. Given a choreography, a compiler produces executable distributed code. In the theory of choreographies, this compilation is called Endpoint Projection (EPP) [1]. A correct EPP has the powerful consequence of guaranteeing deadlock-freedom “for free”: it is syntactically impossible to specify mismatched communication actions in choreographies, so the resulting distributed code cannot get stuck (deadlock-freedom by design) [2].

Recently, there have been two attempts at developing theories that combine the paradigms of choreographic and functional programming, in the hope of reaping the benefits of both [18, 6]. Finding an adequate notion of EPP in this setting has been an issue. In [6] the λ -calculus is extended with choreographic primitives for communications, yielding a simple yet expressive model called Chor λ , but no EPP is presented. In [18] an EPP is given for a choreographic language that extends a standard imperative choreographic language with primitives for abstraction and application (for higher-order composition). However, this theory comes at two important costs when compared to the expected properties of choreographic programming [24]. First, EPP is not modular: changing a part of a choreography that involves only some participants can change also the code projected for other participants. This means

that updating a choreography requires reprojecting and redeploying the entire system, which is not necessary in previous work. Second, participants perform more synchronisations than those written in the choreography. This breaks the design principle that all communications are made syntactically manifest in choreographies.

These issues are not consequences of careless work. Rather, we find that they are both caused by a novel challenge that arises precisely from the combination of functional and choreographic programming – explained in the next paragraph. The aim of this work is to develop a new theory that overcomes this challenge.

The problem

When projecting a choreography to a participant, say *Alice*, the parts of the choreography not involving *Alice* should be ignored [1, 24]. Doing this is simple with traditional imperative choreographies, which are essentially sequences of commands $(c_1; c_2; \dots)$. For each command: if the participant that we are projecting for is involved, we return some (appropriate) code; otherwise, we just skip the command and go to the next. For example, given the choreography $\text{Carol} \rightarrow \text{Bob} : M; \text{Alice} \rightarrow \text{Bob} : M'$, a standard EPP would produce for *Alice* only the code to execute the second command (a send action towards *Bob*).

In a higher-order functional setting, checking if a participant is “involved” in a choreographic term is not an easy syntactic check anymore. Consider a choreography C that takes another choreography x as parameter, runs it, and communicates the result from *Alice* to *Bob*. Since x can be an arbitrary choreography, the participants involved in C are known only after x is instantiated. This is the technical issue that makes defining EPP for functional choreographies nontrivial. In [18], the proposed solution sacrifices modularity: every function application is projected to all participants, who then have to perform a global system synchronisation for every function call.

This work

We define a notion of EPP for $\text{Chor}\lambda$, capitalising on the design of its type system and semantics.

We start our development by focusing on the finite fragment $\text{Chor}\lambda$, i.e., without recursion. First, we introduce a target language for representing distributed code: a distributed λ -calculus, which consists of well-known terms extended with primitives for sending and receiving messages. Then, we use this language to define a modular EPP for (finite) $\text{Chor}\lambda$. The key insight for achieving modularity is the inclusion of a no-op term in the target language, which is the projection of any choreographic term in which a participant is not involved. In this way, if some choreographic subterm does not involve a participant p , it is projected as no-op. And if this term is later edited without involving p , then the projection for p remains no-op and does not need to be recompiled. This is explained in detail in Example 6.

The rule for generating no-ops benefits from the careful design of the rule for typing abstractions in $\text{Chor}\lambda$. This is not an accident: in [6] this particular rule was claimed to be designed with the future development of a suitable EPP in mind, but this was not substantiated. In this paper we show that our EPP satisfies the expected operational correspondence between choreographies and their projections (Theorems 25 and 26). As a consequence, projections of choreographies cannot deadlock.

Furthermore, we define a type system for the target language based on standard techniques, and show that well-typed choreographies are projected onto well-typed target terms whose types are projections of the source choreographic types (Proposition 10). This result is

relevant for applicability: knowing the type of projected functions lets programmers compose them in larger projects through APIs under the control of the programmer, as is commonly done with projected code [15, 17].

A unique feature of $\text{Chor}\lambda$ is that conditionals can use whole choreographies as conditions, and in particular ones that return distributed data structures – data structures that compose data residing at different participants. For the first time, our EPP leverages this feature to offer a new method for capturing *knowledge of choice* – distributed agreement regarding choices between alternative choreographic behaviours [4]. Specifically, we can statically guarantee that two (or more) participants will agree on the instantiation of a sum type (representing alternative choices) solely by performing independent local checks. When this is used in a conditional, it means that all participants are guaranteed to make the same choice at runtime. This gives a simpler alternative to existing verification methods for distributed choices [21]. We call types used in this way *distributed choice types*.

Lastly, we extend our development to the full language of $\text{Chor}\lambda$, including recursion. Recursion allows for divergent behaviour, which gives an interesting problem: a divergent term does not necessarily involve all participants, so generalising the operational correspondence between choreographies and their projections requires allowing choreographies to perform actions involving participants that are not blocked by divergent computations. The semantics of $\text{Chor}\lambda$ include rules for performing reductions out of order, which again were designed with the future development of EPP in mind. We show that these rules are adequate to generalise our results.

Contribution

We define the first notion of EPP for a functional choreographic programming language that is modular and does not add extra communications. This necessitates using not only the information contained in the syntactic structure of a choreography, but also the one contained in the typing derivation that accompanies it. These sources of information give a number of cases for projection that need to be designed carefully, in order to distinguish correctly when a process is potentially involved in the realisation of part of a choreography. We show that EPP satisfies the usual operational correspondence property between choreographies and their projections. Our development also proves two unsubstantiated claims from [6]: that the typing system of $\text{Chor}\lambda$ is expressive enough to support a modular notion of EPP, and that the semantics of $\text{Chor}\lambda$ capture how distributed participants behave in the presence of divergence. Furthermore, we check the practical applicability of our theory by using it to project the model of the Extensible Authentication Protocol (EAP) [28] given in [6], a nontrivial choreography that makes use of higher-order composition, distributed data structures, and distributed choice types.

We anticipate that our developments on the theory of higher-order choreographies will allow higher-order functions to be added to implementations of existing choreographic and similar languages. We discuss this in Section 7.

Structure

We provide a review of the main features of recursion-free $\text{Chor}\lambda$ in Section 2. In Section 3 we describe the local endpoint language $\text{Chor}\lambda$ is projected to and how to project a choreography. We reintroduce recursion into $\text{Chor}\lambda$ and introduce it to our endpoint language in Section 4. An example of a realistic use case (the Extensible Authentication Protocol) projected using our method can be seen in Section 5. Related work is given in Section 6. Conclusions are presented in Section 7. Full definitions and proofs of results for the full language of $\text{Chor}\lambda$ can be found in Appendix A.

2 Background

In this section, we recap the theory of the choreographic λ -calculus (Chor λ) without recursion, from [6]. Chor λ extends the simply typed λ -calculus [5] with primitives that make distribution and communication syntactically manifest.

System model

Chor λ is used to model systems of independent processes, which can interact by synchronous communication. Each process has a name, and knows the names of the other processes in the network. There are two kinds of messages that can be exchanged: *values* are results of computations; and *selection labels* are special constants used to implement agreement on choices about alternative distributed behaviour.

Syntax

The syntax of Chor λ is given by the following grammar

$$\begin{aligned} M &::= V \mid M M \mid \mathbf{case} M \mathbf{of} \mathbf{Inl} x \Rightarrow M; \mathbf{Inr} x \Rightarrow M \mid \mathbf{select}_{\mathbf{p},\mathbf{p}} l M \\ V &::= x \mid \lambda x : T.M \mid \mathbf{Inl} V \mid \mathbf{Inr} V \mid \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{Pair} V V \mid ()@_{\mathbf{p}} \mid \mathbf{com}_{\mathbf{p},\mathbf{p}} \\ T &::= T \rightarrow_{\rho} T \mid T + T \mid T \times T \mid ()@_{\mathbf{p}} \end{aligned}$$

where M is a choreography, V is a value, T is a type, x is a variable, l is a label, \mathbf{p} is a process name, and ρ is a set of process names.

Terms are located at processes, to reflect distribution. For example, the value $()@_{\mathbf{A}}$ reads “the unit value at \mathbf{A} ”. Types are annotated with process names, as well. In the typing rules of Chor λ (shown later), term $()@_{\mathbf{A}}$ has the type $()@_{\mathbf{A}}$, read “the unit type at \mathbf{A} ”. In our examples, for simplicity, we assume the presence of primitives for integer values and an integer type $\mathbf{Int}@_{\mathbf{p}}$ (“an integer at \mathbf{p} ”) – the formal treatment of these are straightforward and similar to that of units.

Abstraction $\lambda x : T.M$, variable x and application MM are as in the standard (simply typed) λ -calculus. Sums and products are constructed, respectively, by using **Inl/Inr** and **Pair**. They are deconstructed in the usual way, respectively with **case** and **fst/snd**. The constructors can take only values as arguments, but this does not restrict expressivity (cf. [6]).

The primitives $\mathbf{com}_{\mathbf{p},\mathbf{q}}$ and $\mathbf{select}_{\mathbf{p},\mathbf{q}} l M$ (where \mathbf{p} and \mathbf{q} are process names) model communications of, respectively, values and selection labels. A *communication* term $\mathbf{com}_{\mathbf{p},\mathbf{q}}$ acts as a function that takes a value at the process named \mathbf{p} and returns the same value at the process named \mathbf{q} . In a *selection* term $\mathbf{select}_{\mathbf{p},\mathbf{q}} l M$, instead, \mathbf{p} informs \mathbf{q} that it has selected the label l before continuing as M . Selections choreographically represent the communication of an internal choice made by \mathbf{p} to \mathbf{q} . As we shall see in our definition of EPP, they play a key role in establishing agreement among processes regarding what behaviour they should enact together.

Selections are standard in choreographic languages and should not to be confused with the distributed choice types that we anticipated in the introduction (these will be illustrated later, in the next section). The former used to implement agreement on choices, whereas the latter are used to codify the information that an agreement has been reached and can thus be used without requiring communication. We will touch on this topic later, in Example 15 and Section 5.

A key feature of $\text{Chor}\lambda$ is distributed data structures. For example, $\mathbf{Pair} ()@p ()@q$ is a distributed pair where the first element resides at p and the second at q . Types record the distribution of values across processes: if p occurs in the type given to V then part of V will be located at p . A function may involve more processes than those listed in the types of its input and output, so the type of abstractions $T \rightarrow_{\rho} T'$ has the extra ingredient ρ , which denotes the processes that may participate in the computation of the function besides those occurring in T or T' . We simply write $T \rightarrow T'$ in place of $T \rightarrow_{\emptyset} T'$. For example, if Alice wants to communicate an integer to Bob directly (without intermediaries), she can use a choreography of type $\text{Int}@Alice \rightarrow \text{Int}@Bob$; however, if the communication might go through a proxy, then she can use a choreography of type $\text{Int}@Alice \rightarrow_{\{\text{Proxy}\}} \text{Int}@Bob$. The information given by ρ gives control on what processes may participate in choreographies taken as arguments. As we show in Section 3, this information is essential to achieve a modular EPP.

We write $\text{fv}(M)$ for the set of free variables in a term M , and $\text{pn}(T)$ and $\text{pn}(M)$ for the set of process names mentioned in respectively a type T and a choreography M . A choreography is *closed* if it has no free variables. Our key results apply to closed choreographies.

► **Example 1** (Remote Function [6]). The following choreography models a distributed computation in which a client, C sends an integer val to a server S and a local function fun located at S is applied to val before the result gets returned to C . The choreography is parametrised on both fun and val .

```
 $\lambda fun : \text{Int}@S \rightarrow_{\emptyset} \text{Int}@S. \lambda val : \text{Int}@C. \mathbf{com}_{S,C} (fun (\mathbf{com}_{C,S} val))$ 
```

Typing

Choreographies are typed with judgements of the form $\Theta; \Gamma \vdash M : T$, where Θ is the set of process names that can be used for typing M and Γ is a function assigning types to variables. We recall a few key typing rules from [6]. Our rules use the notation $\text{pn}(T)$ for the process names that appear in the type T .

$$\frac{\text{pn}(T) = \{p\} \quad \{p, q\} \subseteq \Theta}{\Theta; \Gamma \vdash \mathbf{com}_{p,q} : T \rightarrow_{\emptyset} T[p := q]} \text{[TCOM]}$$

$$\frac{\Theta; \Gamma \vdash N : T \rightarrow_{\rho} T' \quad \Theta; \Gamma \vdash M : T}{\Theta; \Gamma \vdash N M : T'} \text{[TAPP]}$$

$$\frac{\Theta'; \Gamma, x : T \vdash M : T' \quad \rho \cup \text{pn}(T) \cup \text{pn}(T') = \Theta' \subseteq \Theta}{\Theta; \Gamma \vdash \lambda x : T.M : T \rightarrow_{\rho} T'} \text{[TABS]}$$

A communication is typed as a function from any type T located entirely at the sender p to the same type moved to the receiver, as long as both process names are in Θ . Application and abstraction are typed similarly to simply-typed λ -calculus, extended with ρ and Θ (whose consistency is checked in rule TABS). Note that ρ and Θ in rule TABS are not necessarily minimal, and it is possible to type, e.g., $\{p, q\}; \emptyset \vdash \lambda x : \text{Int}@p.x : \text{Int}@p \rightarrow_{\{q\}} \text{Int}@p$. A minimal ρ would consist of those processes that appear either in M or in the types of the free variables of M according to Γ .

► **Example 2.** Let h be the function $\lambda x : \text{Int}@Alice. \mathbf{com}_{\text{Proxy}, \text{Bob}} (\mathbf{com}_{\text{Alice}, \text{Proxy}} x)$, which communicates an integer from Alice to Bob by passing through an intermediary Proxy. Then, $\{Alice, Bob, Proxy\}; \emptyset \vdash h : \text{Int}@Alice \rightarrow_{\{\text{Proxy}\}} \text{Int}@Bob$. For any term M , the composition

h M is well-typed if M has type $\text{Int}@Alice$, denoting that the evaluation of M will yield an integer at Alice. By contrast, h $5@Bob$ is ill-typed because of wrong data locality (the argument is not at the process expected by h).

Semantics

$\text{Chor}\lambda$ comes with an operational semantics given in terms of labelled reductions. Reduction labels are used to keep track of which processes interact in a reduction, which is going to be important for our development. We illustrate this with the two key rules below.

$$\frac{}{\lambda x : T.M \ V \xrightarrow{\emptyset} M[x := V]} \text{[APPABS]} \quad \frac{\text{fv}(V) = \emptyset}{\mathbf{com}_{q,p} \ V \xrightarrow{\{q,p\}} V[q \mapsto p]} \text{[COM]}$$

Rule APPABS is the standard application rule of call-by-value λ -calculus – annotated with an empty set, which indicates that no synchronisation is taking place. Rule COM, instead, implements a communication by “moving” the communicated value from the sender to the receiver (through a substitution). Thus, for example, $\mathbf{com}_{\text{Alice},\text{Bob}} 3@Alice \xrightarrow{\{\text{Alice},\text{Bob}\}} 3@Bob$. Since it makes no sense to communicate a variable whose value is stored at the sender rather than the value itself, we require that the communicated value has no free variables. Communicating a free variable would cause problems for $\text{Chor}\lambda$'s type system, since it would require changing the type of the variable in the environment.

Reductions are labelled with the processes synchronising in them, but this only becomes relevant information in Section 4.

3 Endpoint Projection (EPP) for finite $\text{Chor}\lambda$

In this section we develop a theory of EPP for finite $\text{Chor}\lambda$.

3.1 Process Language

We write implementations of choreographies in a distributed λ -calculus, which we call process language. Processes run in parallel, each with its own behaviour, and can interact by message passing.

Syntax

The syntax of process behaviours is given by the following grammar

$$\begin{aligned} B &::= L \mid B \ B \mid \mathbf{case} \ B \ \mathbf{of} \ \mathbf{Inl} \ x \Rightarrow B; \ \mathbf{Inr} \ x \Rightarrow B \mid \oplus_p \ l \ B \\ &\quad \mid \&_p \{l_1 : B_1, \dots, l_n : B_n\} \\ L &::= x \mid \lambda x : T.B \mid \mathbf{Inl} \ L \mid \mathbf{Inr} \ L \mid \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{Pair} \ L \ L \mid () \mid \mathbf{recv}_p \mid \mathbf{send}_p \mid \perp \\ T &::= T \rightarrow T \mid T + T \mid T \times T \mid () \mid \perp \end{aligned}$$

where B is a behaviour, L is a local value, and T is a local type.

The terms from the λ -calculus are standard. Pairs and sums work as described for $\text{Chor}\lambda$, but note that now they are completely local (as usual) because there are no process name annotations anymore.

The terms for message passing are the local counterparts of choreographic communication terms. Selections are implemented by the *offer* branching term $\&_p \{l_1 : B_1, \dots, l_n : B_n\}$, which offers a number of different ways it can continue for another process p to choose from,

$$\begin{array}{c}
\frac{\Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash \oplus_p l B : T} [\text{NTCHOR}] \quad \frac{\Sigma; \Gamma \vdash B_i : T \text{ for } 1 \leq i \leq n}{\Sigma; \Gamma \vdash \&_p \{l_1 : B_1, \dots, l_n : B_n\} : T} [\text{NTOFF}] \\
\frac{}{\Sigma; \Gamma \vdash \mathbf{send}_p : T \rightarrow \perp} [\text{NTSEND}] \quad \frac{}{\Sigma; \Gamma \vdash \mathbf{recv}_p : \perp \rightarrow T} [\text{NTRECV}] \\
\frac{}{\Sigma; \Gamma \vdash \perp : \perp} [\text{NTBOTM}] \quad \frac{\Sigma; \Gamma \vdash B : \perp \quad \Sigma; \Gamma \vdash B' : \perp}{\Sigma; \Gamma \vdash B B' : \perp} [\text{NTAPP2}]
\end{array}$$

■ **Figure 2** Typing rules for behaviours (selected rules).

and the *choice* term $\oplus_p l B$, which directs p to continue as the behaviour labelled l . Likewise, value communication is divided into a *send* to p action, \mathbf{send}_p , and a *receive* from p action, \mathbf{recv}_p .

We also add the no-op term mentioned in the introduction, \perp , and its type, \perp . A term \perp represents a terminated behaviour with no result. This term is used in the semantics of send and receive: locally, \mathbf{send}_p acts as a function that can take any input and returns \perp , and \mathbf{recv}_p a function that given \perp returns some value. More interestingly, \perp also plays an important role wrt modularity in our notion of EPP, which we will discuss later in our presentation of projection. All types but \perp are standard (as in $\text{Chor}\lambda$, but without process name annotations).

A system of running processes is called a *network*.

► **Definition 3.** A network \mathcal{N} is a finite map from a set of process names to behaviours.

Given two networks \mathcal{N} and \mathcal{N}' with disjoint domains, their parallel composition $\mathcal{N} \mid \mathcal{N}'$ maps each process name to the behaviour in the network defining the process. Any network is equivalent to a parallel composition of networks with singleton domains, so we write $p_1[B_1] \mid \dots \mid p_n[B_n]$ for the network where each process p_i has behaviour B_i [24].

► **Example 4.** Consider the choreography $\mathbf{com}_{B,C} (\mathbf{com}_{A,B} ()@A)$. A correct implementation is the network $A[\mathbf{send}_B ()] \mid B[\mathbf{send}_C (\mathbf{recv}_A \perp)] \mid C[\mathbf{recv}_B \perp]$.

Typing

Behaviours are typed with judgements of the form $\Gamma \vdash B : T$. The typing rules are the local counterparts of those in $\text{Chor}\lambda$, obtained by removing Θ and process names in types. We add the \perp type for terms that can result in \perp . Figure 2 displays representative typing rules to deal with \perp and communications.

Semantics

The semantics of networks is given as a labelled transition system. Figure 3 displays some representative transition rules.

Labels for network transitions have the form τ_P , where P ranges over sets of one or two process names. Rule NPRO annotates an internal transition by a process with its name, and rule NPAR lifts transitions in parallel compositions.

The transition axioms for send and receive are typical of process calculi with early semantics. Send and receive transitions are matched in rule NCOM to perform a communication. The label $\tau_{p,q}$ denotes an internal move (τ) and manifests the names of processes that contribute to performing it (p and q). We treat the subscript p, q as an unordered set that consists of the two process names.

$$\begin{array}{c}
\frac{fv(L) = \emptyset}{\text{send}_p L \xrightarrow{\text{send}_p L} \perp} \text{[NSEND]} \quad \frac{}{\text{rcv}_p \perp \xrightarrow{\text{rcv}_p L} L} \text{[NRECV]} \\
\frac{B_1 \xrightarrow{\text{send}_q L} B'_1 \quad B_2 \xrightarrow{\text{rcv}_p L} B'_2}{p[B_1] \mid q[B_2] \xrightarrow{\tau_{p,q}} p[B'_1] \mid q[B'_2]} \text{[NCOM]} \\
\frac{}{\oplus_p l B \xrightarrow{\oplus_p l} B} \text{[NCHO]} \quad \frac{}{\&_p \{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\&_p \ell_i} B_i} \text{[NOFF]} \\
\frac{B_1 \xrightarrow{\oplus_q \ell} B'_1 \quad B_2 \xrightarrow{\&_p \ell} B'_2}{p[B_1] \mid q[B_2] \xrightarrow{\tau_{p,q}} p[B'_1] \mid q[B'_2]} \text{[NSEL]} \\
\frac{}{(\lambda x : T.B) L \xrightarrow{\tau} B[x := L]} \text{[NABSAPP]} \quad \frac{}{\perp \perp \xrightarrow{\tau} \perp} \text{[NBOTM]} \\
\frac{B \xrightarrow{\tau} B'}{p[B] \xrightarrow{\tau_p} p[B']} \text{[NPRO]} \quad \frac{\mathcal{N} \xrightarrow{\tau_p} \mathcal{N}''}{\mathcal{N} \mid \mathcal{N}' \xrightarrow{\tau_p} \mathcal{N}'' \mid \mathcal{N}'} \text{[NPAR]}
\end{array}$$

■ **Figure 3** Network semantics (representative rules).

The P-annotations in labels enable the formulation of the next lemma, which we use in some of our proofs to focus on the processes involved in a transition. The proof of this result and others for the full Chor λ language are provided in Appendix A.

► **Lemma 5.** *For any p and \mathcal{N} , if $\mathcal{N} \xrightarrow{\tau_p} \mathcal{N}'$ and $p \notin P$ then $\mathcal{N}(p) = \mathcal{N}'(p)$.*

Most of the other rules follow the same intuition and are otherwise standard. The exception is rule NBOTM, which garbage collects \perp terms. We discuss the role of this rule in Example 9, after having presented our notion of EPP.

3.2 Endpoint Projection (EPP)

We now move to defining the endpoint projection (EPP) of a choreography M for an individual process p , assuming that M is well-typed; that is, $\Theta; \Gamma \vdash M : T$ for some Θ , Γ , and T . The definition of EPP formally depends on this typing derivation, but to keep notation simple we write just $\llbracket M \rrbracket_p$ for the projection of M on p and refer to the type T associated to M in the specific derivation we are looking at as $\text{type}(M)$.

Projection translates each choreographic term to a corresponding local behaviour. For example, a communication term $\mathbf{com}_{p,q}$ is projected to a send action for the sender p and a receive action for the receiver q .

Abstraction presents a novel challenge compared to previous, non-functional choreographic languages. We discuss it in the next example, which also illustrates the importance of \perp in our theory of EPP.

► **Example 6.** Let $M = \lambda x : \text{Int}@p.M'$ for some M' , and consider the issue of defining its projection on a process q different than p , $\llbracket M \rrbracket_q$. Since EPP is usually defined inductively on the structure of the choreography, this definition should not depend on the context that M is used in.

The standard principle for EPP found in the literature is to ignore the parts that do not mention the process we are currently projecting to. Following this principle, we should omit the initial abstraction (λx) of M in the implementation of q .

7:10 Modular Compilation for Higher-Order Functional Choreographies

For example, for $M = \lambda x : \text{Int}@p.2@q$, we could design EPP such that $\llbracket M \rrbracket_q = 2$. This works when M is used in an application as $(\lambda x : \text{Int}@p.2@q) 1@p$, where $\llbracket M \rrbracket_q = 2$ is still reasonable (since q has nothing to do with the argument).

Unfortunately, this standard approach is not robust in the case of functional choreographies: even if q is not mentioned in the type of x in $\lambda x : \text{Int}@p$, in general it could still participate in the context that produces the value that x is going to be replaced with. For example, let $M'' = (\lambda x : \text{Int}@p.\text{com}_{q,p} 2@q) (\text{com}_{q,p} 1@q)$, which expresses a sequence of communications between q and p (first of 1 and then of 2, in order). If we insist on excluding the abstraction from the projection on q , then we obtain $\llbracket M'' \rrbracket_q = (\text{send}_p 2) (\text{send}_p 1)$. This is wrong, because it would send 2 before 1. Therefore, we cannot just skip abstractions that do not involve the process we are projecting on. In this case, a correct implementation of q in M'' would be $(\lambda x : \perp.\text{send}_p 2) (\text{send}_p 1)$. Our process language is carefully designed to make terms like this normalise gracefully: after executing $\text{send}_p 1$ the righthandside is \perp , thus allowing for the application to be resolved and for the second send action to be executed.

Sometimes, however, abstractions should be skipped. For example, if M is $\lambda x : \text{Int}@p.1@p$, then $\llbracket M \rrbracket_q$ should clearly be \perp . The alternative, $\lambda x : \perp.\perp$, would break modularity of EPP because the structure of $\llbracket M \rrbracket_q$ would depend on the internal behaviour of p . To solve this issue, we take the approach of skipping an abstraction like $\lambda x : T.M'$ only if both T and M' do not mention the process that we are projecting on. Type information is therefore key to our EPP, in addition to the usual syntactic checks, which is why we have made the EPP dependent on a typing derivation.

We will come back to \perp and its companion rule NBOTM in Example 9. J

In order to define EPP precisely, we need a few additional ingredients.

Projecting a term M requires knowing the processes involved in its type. As our EPP takes an entire typing derivation of M as input, the type is implicitly given in the derivation provided to EPP. So we write without ambiguity $\text{pn}(\text{type}(M))$ for this set of process names.

The second ingredient concerns knowledge of choice. When projecting a conditional **case** M of **Inl** $x \Rightarrow M'$; **Inr** $y \Rightarrow M''$, processes not occurring in M cannot know what branch of the choreography is chosen; therefore, the projections of M' and M'' must be combined in a uniquely-defined behaviour. We thus define a partial *merge* operator (\sqcup), adapted from [1, 8, 19], whose key property is

$$\&\{l_i : B_i\}_{i \in I} \sqcup \&\{l_j : B'_j\}_{j \in J} = \&\{l_k : B_k \sqcup B'_k\}_{k \in I \cap J} \cup \{l_i : B_i\}_{i \in I \setminus J} \cup \{l_j : B'_j\}_{j \in J \setminus I}$$

and which is homomorphically defined for the remaining constructs (see Appendix A for the full definition). The idea is that a process not in M must either perform the same actions in M' and M'' (so the choice does not matter) or receive an appropriate selection to know which branch has been chosen. Merging of incompatible behaviours is undefined.

► **Example 7.** Consider the choreography

$$C = \text{case Inl } ()@p \text{ of Inl } x \Rightarrow \text{select}_{p,q} \text{ left } 0@q; \text{ Inr } y \Rightarrow \text{select}_{p,q} \text{ right } 1@q.$$

Using merging, its projection on process q is $\llbracket C \rrbracket_q = \&_p\{\text{left} : 0, \text{right} : 1\}$. J

► **Definition 8.** The EPP of a choreography M on a specific process p ($\llbracket M \rrbracket_p$) is defined by the rules in Figure 4. The EPP of a choreography ($\llbracket M \rrbracket$) is the parallel composition of the EPPs on its processes: $\llbracket M \rrbracket = \prod_{p \in \text{pn}(M)} p \llbracket M \rrbracket_p$.

Intuitively, projecting a choreography on a process that is not involved in it returns a \perp . In general, however, a choreography may involve processes not mentioned in its type. This explains the first clause for projecting an application: even if p does not appear in the type of

Choreographies

$$\begin{aligned}
\llbracket M \ N \rrbracket_{\mathfrak{p}} &= \begin{cases} \llbracket M \rrbracket_{\mathfrak{p}} \ \llbracket N \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(M)) \text{ or } \mathfrak{p} \in \text{pn}(M) \cap \text{pn}(N) \\ \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \llbracket N \rrbracket_{\mathfrak{p}} = \perp \\ \llbracket N \rrbracket_{\mathfrak{p}} & \text{otherwise} \end{cases} \\
\llbracket \lambda x : T.M \rrbracket_{\mathfrak{p}} &= \begin{cases} \lambda x : \llbracket T \rrbracket_{\mathfrak{p}} . \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\lambda x : T.M)) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{case } M \text{ of } \mathbf{Inl} \ x \Rightarrow N; \mathbf{Inr} \ x' \Rightarrow N' \rrbracket_{\mathfrak{p}} &= \\
&\begin{cases} \text{case } \llbracket M \rrbracket_{\mathfrak{p}} \text{ of } \mathbf{Inl} \ x \Rightarrow \llbracket N \rrbracket_{\mathfrak{p}}; \mathbf{Inr} \ x' \Rightarrow \llbracket N' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(M)) \\ \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \llbracket N \rrbracket_{\mathfrak{p}} = \llbracket N' \rrbracket_{\mathfrak{p}} = \perp \\ \llbracket N \rrbracket_{\mathfrak{p}} \sqcup \llbracket N' \rrbracket_{\mathfrak{p}} & \text{if } \llbracket M \rrbracket_{\mathfrak{p}} = \perp \\ (\lambda x'' : \perp . \llbracket N \rrbracket_{\mathfrak{p}} \sqcup \llbracket N' \rrbracket_{\mathfrak{p}}) \ \llbracket M \rrbracket_{\mathfrak{p}} & \text{otherwise, for some} \\ & \quad x'' \notin \text{fv}(N) \cup \text{fv}(N') \end{cases} \\
\llbracket \mathbf{Inl} \ V \rrbracket_{\mathfrak{p}} &= \begin{cases} \mathbf{Inl} \ \llbracket V \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\mathbf{Inl} \ V)) \\ \perp & \text{otherwise} \end{cases} \quad \llbracket \mathbf{fst} \rrbracket_{\mathfrak{p}} = \begin{cases} \mathbf{fst} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\mathbf{fst})) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{select}_{\mathfrak{q}, \mathfrak{q}'} \ l \ M \rrbracket_{\mathfrak{p}} &= \begin{cases} \oplus_{\mathfrak{q}'} \ l \ \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} = \mathfrak{q} \neq \mathfrak{q}' \\ \&_{\mathfrak{q}} \{l : \llbracket M \rrbracket_{\mathfrak{p}}\} & \text{if } \mathfrak{p} = \mathfrak{q}' \neq \mathfrak{q} \\ \llbracket M \rrbracket_{\mathfrak{p}} & \text{otherwise} \end{cases} \\
\llbracket \text{com}_{\mathfrak{q}, \mathfrak{q}'} \rrbracket_{\mathfrak{p}} &= \begin{cases} \lambda x : \llbracket T \rrbracket_{\mathfrak{p}} . x & \text{if } \mathfrak{p} = \mathfrak{q} = \mathfrak{q}' \text{ and } \text{type}(\text{com}_{\mathfrak{q}, \mathfrak{q}'}) = T \rightarrow_{\emptyset} T' \\ \text{send}_{\mathfrak{q}'} & \text{if } \mathfrak{p} = \mathfrak{q} \neq \mathfrak{q}' \\ \text{recv}_{\mathfrak{q}} & \text{if } \mathfrak{p} = \mathfrak{q}' \neq \mathfrak{q} \\ \perp & \text{otherwise} \end{cases} \\
\llbracket () @ \mathfrak{q} \rrbracket_{\mathfrak{p}} &= \begin{cases} () & \text{if } \mathfrak{q} = \mathfrak{p} \\ \perp & \text{otherwise} \end{cases} \quad \llbracket x \rrbracket_{\mathfrak{p}} = \begin{cases} x & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(x)) \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

Types

$$\begin{aligned}
\llbracket () @ \mathfrak{q} \rrbracket_{\mathfrak{p}} &= \begin{cases} () & \text{if } \mathfrak{q} = \mathfrak{p} \\ \perp & \text{otherwise} \end{cases} \quad \llbracket T \times T' \rrbracket_{\mathfrak{p}} = \begin{cases} \llbracket T \rrbracket_{\mathfrak{p}} \times \llbracket T' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(T \times T') \\ \perp & \text{otherwise} \end{cases} \\
\llbracket T \rightarrow_{\rho} T' \rrbracket_{\mathfrak{p}} &= \begin{cases} \llbracket T \rrbracket_{\mathfrak{p}} \rightarrow \llbracket T' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \rho \cup \text{pn}(T) \cup \text{pn}(T') \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

■ **Figure 4** Projecting a choreography in Chor λ onto a process – when cases overlap, the first one takes precedence (representative rules).

M , it may participate in interactions in M . Vice versa, a process can appear in the type of a choreography without appearing in the choreography itself. The difference between a process appearing in a choreography or its type becomes important when we look at the projection of **case** M of **lnl** $x \Rightarrow N$; **lnr** $x' \Rightarrow N'$. Here, \mathfrak{p} appearing in the type of M indicates that \mathfrak{p} will, at the end of the computation of M , know what branch will be chosen; therefore, the projection on \mathfrak{p} is a **case**. However, it is possible that \mathfrak{p} is involved in the computation of the condition M without knowing the final choice, e.g., if $M = \mathbf{com}_{\mathfrak{p},\mathfrak{q}} M'$. In this case, the projection on \mathfrak{p} is not a **case** but still needs code to participate in the implementation of M correctly. If \mathfrak{p} is involved in the branches as well, then we need to project code for them too: we inject an abstraction in order to maintain the correct order of computation (M before N and N') and make the resulting process well typed (since \mathfrak{p} does not appear in the type of M , that type will be projected to \perp).

The projection of abstraction illustrates the necessity of the ρ annotation on abstraction types. For example, consider an application of a communication via a proxy ($\lambda x : \text{Int}@_{\mathfrak{p}} \rightarrow_{\{\mathfrak{r}\}} \text{Int}@_{\mathfrak{q}}.x \text{ } 3@_{\mathfrak{p}} \text{ } (\lambda y : \text{Int}@_{\mathfrak{p}}.\mathbf{com}_{\mathfrak{r},\mathfrak{q}} \mathbf{com}_{\mathfrak{p},\mathfrak{r}} y)$). Without the annotation $\{\mathfrak{r}\}$ in subterm ($\lambda x : \text{Int}@_{\mathfrak{p}} \rightarrow_{\{\mathfrak{r}\}} \text{Int}@_{\mathfrak{q}}.x \text{ } 3@_{\mathfrak{p}} \text{ } (\lambda y : \text{Int}@_{\mathfrak{p}}.\mathbf{com}_{\mathfrak{r},\mathfrak{q}} \mathbf{com}_{\mathfrak{p},\mathfrak{r}} y)$), the projection of this subterm on \mathfrak{r} would just be \perp , which is wrong for the overall application since \mathfrak{r} will actually be involved.

Selections and communications follow the intuition given before, with one interesting detail: self-selections are ignored, and self-communications are projected to the identity function. This is different from previous works, where self-communication is not allowed – here we lift this restriction.

Likewise, projecting a type T yields \perp at any process not used in T .

► **Example 9.** Let $M = (\mathbf{com}_{\mathfrak{p},\mathfrak{q}} (\lambda x : \text{Int}@_{\mathfrak{p}}.3@_{\mathfrak{p}})) (\mathbf{com}_{\mathfrak{p},\mathfrak{q}} 5@_{\mathfrak{p}})$, where a function and a value are both sent from \mathfrak{p} to \mathfrak{q} before being applied at \mathfrak{q} . The implementation of \mathfrak{q} is $\llbracket M \rrbracket_{\mathfrak{q}} = (\mathbf{recv}_{\mathfrak{p}} \perp) (\mathbf{recv}_{\mathfrak{p}} \perp)$, whose execution is straightforward. At \mathfrak{p} , however, we have that $\llbracket M \rrbracket_{\mathfrak{p}} = (\mathbf{send}_{\mathfrak{q}}(\lambda x : \text{Int}.3)) (\mathbf{send}_{\mathfrak{q}} 5)$, which after executing the two send actions becomes $\perp \perp$. After executing its two communications, the choreography M becomes $M' = (\lambda x : \text{Int}@_{\mathfrak{q}}.3@_{\mathfrak{q}}) 5@_{\mathfrak{q}}$. M' is located entirely at \mathfrak{q} , and therefore $\llbracket M' \rrbracket_{\mathfrak{p}} = \perp$, which is different than the $\perp \perp$ reached by $\llbracket M \rrbracket_{\mathfrak{p}}$. We therefore need a way to make the application $\perp \perp$ become \perp . Rule NBTM serves this purpose. The fact that this is not possible with two units is the key semantic difference between \perp and $()$. J

► **Proposition 10.** *Let M be a closed choreography. If $\Theta; \Gamma \vdash M : T$, then for any process \mathfrak{p} appearing in M , we have that $\llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket_{\mathfrak{p}} : \llbracket T \rrbracket_{\mathfrak{p}}$, where $\llbracket \Gamma \rrbracket$ are defined by applying EPP to all types occurring Γ .*

► **Example 11.** Let M be the remote function choreography in Example 1. Its projections on \mathfrak{C} and \mathfrak{S} are as follows.

$$\llbracket M \rrbracket_{\mathfrak{C}} = \lambda f : \perp. \lambda val : \text{Int}. \mathbf{recv}_{\mathfrak{S}} (\mathbf{send}_{\mathfrak{S}} val)$$

$$\llbracket M \rrbracket_{\mathfrak{S}} = \lambda f : (\text{Int} \rightarrow \text{Int}). \lambda val : \perp. \mathbf{send}_{\mathfrak{C}} (f (\mathbf{recv}_{\mathfrak{C}} \perp))$$

This example illustrates the key features discussed in the text: projection of communications as two dual actions; and the way function applications are projected when the process does not appear in the function's type. J

We describe what we consider modularity of EPP, formally defined in Definition 12. Modular projection means that for any context $C[\]$ the projection of $C[M]$ at \mathfrak{p} will be the same for any M which does not involve \mathfrak{p} . The definition of context is as expected and can be found in Appendix A. Modularity is typical (and expected) of EPP, because the projection of \mathfrak{p} should not be generating junk code based on the behaviour of other processes.

► **Definition 12** (Modularity of EPP). *An EPP $\llbracket - \rrbracket$ is called modular if $\llbracket C[M] \rrbracket_{\mathfrak{p}} = \llbracket C[N] \rrbracket_{\mathfrak{p}}$ for any process \mathfrak{p} , context $C[\]$, and choreographies M and N such that $\Theta; \Gamma \vdash M : T$ and $\Theta; \Gamma \vdash N : T$ with $\mathfrak{p} \notin \Theta$.*

Modularity ensures that if we modify part of a choreography in which a process \mathfrak{p} is not involved, we do not need to recompile the projection of the choreography onto \mathfrak{p} because this projection is unaffected. In general, the strong equality requirement could be relaxed to allow for some extra local actions that do not change the observable behaviour of a process, e.g., adding “empty” applications like $\lambda x. \perp : \perp$. This would yield some extra flexibility to deal with cases such as the one seen in Example 6, so long as the interactions with other processes and return value at \mathfrak{p} do not change. However, this design would come at some costs: an increase in complexity due to the addition of a suitable notion of behavioural equivalence; a potential loss in efficiency, since processes might gain unnecessary reductions in their projections; and a potential leak of information, since the local code projected on a process would reveal some information about the behaviours of other processes.

The following proposition, Proposition 14, shows that our EPP is modular.

► **Lemma 13.** *Given a choreography M , if $\Theta; \Gamma \vdash M : T$ and $\mathfrak{p} \notin \Theta$ then $\llbracket M \rrbracket_{\mathfrak{p}} = \perp$.*

Proof. Follows from $\mathfrak{p} \notin \Theta$ implying $\mathfrak{p} \notin \text{pn}(T) \cup \text{pn}(M)$ and induction on the derivation of $\llbracket M \rrbracket_{\mathfrak{p}}$. ◀

► **Proposition 14.** *The EPP $\llbracket - \rrbracket$ given in Definition 8 is modular.*

Proof. Follows from Lemma 13 and observing that the projection of any context always treats \perp the same. ◀

► **Example 15** (Distributed choice types). Now that we can project a choreography, we return to the idea of distributed choice types from the introduction. Consider a choreography

$$M = \lambda x : \text{Bool}@(\mathfrak{p}, \mathfrak{q}). \text{case } x \text{ of } \mathbf{Inl } y \Rightarrow \mathbf{comp}_{\mathfrak{p}, \mathfrak{q}} 3 @ \mathfrak{p}; \mathbf{Inr } y \Rightarrow 5 @ \mathfrak{q}$$

Here $\text{Bool}@(\mathfrak{p}, \mathfrak{q})$ is equivalent to the type $(()@_{\mathfrak{p}} \times ()@_{\mathfrak{q}}) + (()@_{\mathfrak{p}} \times ()@_{\mathfrak{q}})$, and in general we can encode a “distributed boolean” as

$$\text{Bool}@_{\vec{\mathfrak{p}}} = (()@_{\mathfrak{p}_1} \times \cdots \times ()@_{\mathfrak{p}_n}) + (()@_{\mathfrak{p}_1} \times \cdots \times ()@_{\mathfrak{p}_n})$$

We can use distributed booleans to codify distributed choices, in this case by having both \mathfrak{p} and \mathfrak{q} be able to make local choice without interacting but still guaranteeing that they choose their respective behaviours correctly.

Specifically, when we project M we get two local choices made at \mathfrak{p} and \mathfrak{q} , both of which are guaranteed to make the same choice. First we have the projections

$$\llbracket M \rrbracket_{\mathfrak{p}} = \lambda x : (() \times \perp) + (() \times \perp). \text{case } x \text{ of } \mathbf{Inl } y \Rightarrow \mathbf{send}_{\mathfrak{q}} 3; \mathbf{Inr } y \Rightarrow \perp$$

and

$$\llbracket M \rrbracket_{\mathfrak{q}} = \lambda x : (\perp \times ()) + (\perp \times []). \text{case } x \text{ of } \mathbf{Inl } y \Rightarrow \mathbf{recv}_{\mathfrak{p}} \perp; \mathbf{Inr } y \Rightarrow 5$$

For these processes to be deadlock-free when put in parallel, we need both of them to make the same choice. Thankfully, the distributed boolean type ensures that x will always be instantiated as either $\mathbf{Inl} (\mathbf{Pair} ()@_{\mathfrak{p}} ()@_{\mathfrak{q}})$ or $\mathbf{Inr} (\mathbf{Pair} ()@_{\mathfrak{p}} ()@_{\mathfrak{q}})$. From the projection we get $\llbracket \mathbf{Inl} (\mathbf{Pair} ()@_{\mathfrak{p}} ()@_{\mathfrak{q}}) \rrbracket_{\mathfrak{p}} = \mathbf{Inl} (\mathbf{Pair} () \perp)$ and $\llbracket \mathbf{Inl} (\mathbf{Pair} ()@_{\mathfrak{p}} ()@_{\mathfrak{q}}) \rrbracket_{\mathfrak{q}} = \mathbf{Inl} (\mathbf{Pair} \perp ())$,

7:14 Modular Compilation for Higher-Order Functional Choreographies

and similar for the **Inr** case. We therefore know that $\text{Chor}\lambda$'s distributed choice works as intended when projected. As we shall see in Section 5, one use for this technique is to have different processes independently agree on the size of a distributed list.

Note that if we tried to model a distributed boolean as $(\text{()@p} + \text{()@p}) \times (\text{()@q} + \text{()@q})$, it would not be useful to represent a distributed choice because it would allow the processes to make different choices. (Also, M would obviously not be well-typed, as a condition must have a sum type.)

We now show that there is a close correspondence between the executions of choreographies and of their projections. Intuitively, this correspondence states that a choreography can execute an action if, and only if, its projection can execute the same action, and both transition to new terms in the same relation. Technically, we need to be more precise: if a choreography M reduces by rule **CASE**, then the result has fewer branches than the network obtained by performing the corresponding reduction in the projection of M . (This is a standard issue with choreographic conditionals [24].)

In order to capture this, we define a partial order \sqsupseteq that relates a behaviour to a version with fewer branches: $B \sqsupseteq B'$ iff $B \sqcup B' = B$. Intuitively, if $B \sqsupseteq B'$, then B offers the same or more branches than B' (also in subterms). This notion extends to networks by defining $\mathcal{N} \sqsupseteq \mathcal{N}'$ to mean that, for any process \mathfrak{p} , $\mathcal{N}(\mathfrak{p}) \sqsupseteq \mathcal{N}'(\mathfrak{p})$. Example 16 shows the necessity of \sqsupseteq in order to get a meaningful notion of operational correspondence between choreographies and their projection.

► **Example 16.** Consider again the choreography from Example 7,

$$C = \mathbf{case} \text{ Inl } (\text{()@p} \text{ of } \text{Inl } x \Rightarrow \mathbf{select}_{\mathfrak{p},\mathfrak{q}} \text{ left } 0\text{@q}; \text{Inr } y \Rightarrow \mathbf{select}_{\mathfrak{p},\mathfrak{q}} \text{ right } 1\text{@q}),$$

and its projection B on \mathfrak{q} , $B = \llbracket C \rrbracket_{\mathfrak{q}} = \&_{\mathfrak{p}}\{\text{left} : 0, \text{right} : 1\}$.

When entering the **case**, C reduces to $C' = \mathbf{select}_{\mathfrak{p},\mathfrak{q}} \text{ left } 0\text{@q}$, but \mathfrak{q} is not involved in this action and its behaviour remains B , which is not $\llbracket C' \rrbracket_{\mathfrak{q}}$. However, $\&_{\mathfrak{p}}\{\text{left} : 0, \text{right} : 1\} \sqcup \&_{\mathfrak{p}}\{\text{left} : 0\} = \&_{\mathfrak{p}}\{\text{left} : 0, \text{right} : 1\}$, so $B \sqsupseteq \llbracket C' \rrbracket_{\mathfrak{q}}$.

In addition to \sqsupseteq , we need to equate behaviours that differ only by applications to \perp like P and $(\lambda x : \perp.P) \perp$ introduced by the projection of applications.

► **Definition 17.** We define \equiv as the least equivalence relation on behaviours that is closed under context and $P \equiv (\lambda x : \perp.P) \perp$ for any behaviour P . We write $\mathcal{N} \equiv \mathcal{N}'$ for the pointwise extension of \equiv to networks (i.e., $\Pi_{\mathfrak{p}}\mathfrak{p}[P_{\mathfrak{p}}] \equiv \Pi_{\mathfrak{p}}\mathfrak{p}[P'_{\mathfrak{p}}]$ iff $P_{\mathfrak{p}} \equiv P'_{\mathfrak{p}}$ for all \mathfrak{p} s) and $\mathcal{N} \sqsupseteq \mathcal{N}'$ if there is a network \mathcal{N}'' such that $\mathcal{N} \sqsupseteq \mathcal{N}''$ and $\mathcal{N}'' \equiv \mathcal{N}'$.

We can finally show that the EPP of a choreography can do all that (completeness) and only what (soundness) the choreography does. Here \rightarrow^* denotes a sequence of transitions with any labels, and \rightarrow^+ a nonempty such sequence.

► **Theorem 18 (Completeness).** Given a closed choreography M , if $M \xrightarrow{P} M'$, $\Theta; \Gamma \vdash M : T$, and $\llbracket M \rrbracket$ is defined, then there exist networks \mathcal{N} and M'' such that: $\llbracket M \rrbracket \rightarrow^+ \mathcal{N}$; $M' \rightarrow^* M''$; and $\mathcal{N} \sqsupseteq \llbracket M'' \rrbracket$.

► **Theorem 19 (Soundness).** Given a closed choreography M , if $\Theta; \Gamma \vdash M : T$ and $\llbracket M \rrbracket \rightarrow^* \mathcal{N}$ for some network \mathcal{N} , then there exist a choreography M' , and a network \mathcal{N}' such that: $M \rightarrow^* M'$; $\mathcal{N} \rightarrow^* \mathcal{N}'$; and $\mathcal{N}' \sqsupseteq \llbracket M' \rrbracket$.

Since we have no recursion and only require that the choreography and projection eventually get to the same state, we can prove soundness and correctness without needing the out-of-order semantics usually required in choreographic languages [24].

From Theorems 18 and 19 and the type preservation and progress results from [6], we obtain deadlock-freedom: the EPP of a well-typed closed choreography can continue to reduce until all processes contain only local values.

► **Corollary 20** (Deadlock-freedom). *Given a closed choreography M , if $\Theta; \Gamma \vdash M : T$ then: whenever $\llbracket M \rrbracket \rightarrow^* \mathcal{N}$ for some network \mathcal{N} , either there exists \mathfrak{p} and \mathcal{N}' such that $\mathcal{N} \xrightarrow{\tau} \mathcal{N}'$ or $\mathcal{N} = \prod_{\mathfrak{p} \in \text{pn}(M)} \mathfrak{p}[L_{\mathfrak{p}}]$.*

4 Recursion

So far we have worked with a recursion-free subset of $\text{Chor}\lambda$. In this section, we extend our development to the full language presented of $\text{Chor}\lambda$, which includes recursive definitions [6]. As we will see, recursion is technically challenging because of the introduction of divergence.

4.1 Definitions

Choreographies

Recursion in $\text{Chor}\lambda$ is achieved by named functions (f) parametrised on process names. We use D to range over mappings of parametrised functions names to choreographies (the bodies of the functions). To execute a choreography M containing calls to named functions, the choreography must be associated with a mapping D that contains all the named functions called by M . The grammar of choreographies is extended with $M ::= \dots \mid f(\vec{\mathfrak{p}})$. A function call $f(\vec{\mathfrak{p}})$ invokes f by instantiating its parameters with the process names $\vec{\mathfrak{p}}$, which evaluates to the body of the function. In a function call or definition, parameters must be distinct. Semantically, we add D as an annotation to the reduction relation for choreographies and use the following rule to evaluate functions. Labels in $\text{Chor}\lambda$ with recursion are extended to the form ℓ, \mathfrak{P} , where the new ingredient ℓ can be either τ or λ . The need for ℓ is explained later.

$$\frac{D(f(\vec{\mathfrak{p}})) = M}{f(\vec{\mathfrak{p}}) \xrightarrow{\tau, \emptyset}_D M[\vec{\mathfrak{p}} \mapsto \vec{\mathfrak{p}}]} \text{ [DEF]}$$

To type recursive choreographies, we introduce recursive type variables ranged over by t . These are defined in a collection Σ , which contains type equations of the form $t@_{\vec{\mathfrak{p}}} = T$ – the elements of $\vec{\mathfrak{p}}$ must be distinct. The grammar of types is extended with parametrised variables: $T ::= \dots \mid t@_{\vec{\mathfrak{p}}}$. Essentially, assuming the presence of an equation $t@_{\vec{\mathfrak{p}'}} = T$, $t@_{\vec{\mathfrak{p}}}$ can be unfolded into $T[\vec{\mathfrak{p}} := \vec{\mathfrak{p}}]$. Typing judgements are then of the form $\Theta; \Sigma; \Gamma \vdash M : T$, where Γ may now also contain type assignments for recursive functions of the form $f(\vec{\mathfrak{p}}) : T$.

$$\frac{\Theta; \Sigma; \Gamma \vdash M : t@_{\vec{\mathfrak{p}}} \quad t@_{\vec{\mathfrak{p}}} =_{\Sigma} T \quad \vec{\mathfrak{p}} \subseteq \Theta \quad \|\vec{\mathfrak{p}}\| = \|\vec{\mathfrak{p}}'\| \quad \vec{\mathfrak{p}}' \text{ distinct}}{\Theta; \Sigma; \Gamma \vdash M : T[\vec{\mathfrak{p}} := \vec{\mathfrak{p}}']} \text{ [TEQ]}$$

We also write $\Theta; \Sigma; \Gamma \vdash D$ to denote that each function in D can be typed accordingly to its type in Γ .

► **Example 21** (Remote Map). With recursive functions, we can write more complex choreographies that call themselves and each other. Let `remoteFunction(C, S)` be defined as the choreography in Example 1. We use it to define a function `remoteMap(C, S)`, where a

7:16 Modular Compilation for Higher-Order Functional Choreographies

server S applies a function to not just one value, but instead to each element of a stream communicated from a client C . Then S returns the results, which C gathers into a list with the standard `cons` function used to construct a new list.

```
remoteMap(C, S) = λfun : Int@S → Int@S. λlist : [Int]@C.
  case list of
  Inl x ⇒ selectC,S stop ()@C;
  Inr x ⇒ selectC,S again
    cons(C) (remoteFunction(C, S) fun (fst x)) (remoteMap(C, S) fun (snd x))
```

Here, $[Int]@C$ is defined as $[Int]@C = ()@C + (Int@C \times [Int]@C)$, representing a list of integers. In general, we write $[t]@(p_1, \dots, p_n)$ to mean the type satisfying $[t]@(p_1, \dots, p_n) = (()@p_1 \times \dots \times ()@p_n) + (t@(p_1, \dots, p_n) \times [t]@(p_1, \dots, p_n))$.

► **Example 22** (Diffie-Hellman [6]). We recall the choreography for the Diffie-Hellman key exchange protocol [13], which allows two processes to agree on a shared secret key without assuming secrecy of communications. Again, we use the primitive type `Int`.

To define this protocol, we use the local function `modPow(R)` of the type

$$\text{modPow}(R) : \text{Int}@R \rightarrow \text{Int}@R \rightarrow \text{Int}@R \rightarrow \text{Int}@R$$

which computes powers with a given modulo. Given `modPow(R)`, we can implement Diffie-Hellman as the following choreography:

```
diffieHellman(P, Q) =
  λpsk : Int@P. λqsk : Int@Q. λpsg : Int@P.
  λqsg : Int@Q. λpsp : Int@P. λqsp : Int@Q.
  pair (modPow(P) psg (comQ,P (modPow(Q) qsg qsk qsp)) psp)
    (modPow(Q) qsg (comP,Q (modPow(P) psg psk psp)) qsp)
```

Given the individual secret keys (psk and qsk) and a previously publicly agreed upon shared prime modulus and base ($psg = qsg, psp = qsp$), the participants exchange their locally-computed public keys in order to arrive at a shared key that can be used to encrypt all further communication. This means `diffieHellman(P, Q)` has the type:

$$\text{Int}@P \rightarrow \text{Int}@Q \rightarrow \text{Int}@P \rightarrow \text{Int}@Q \rightarrow \text{Int}@P \rightarrow \text{Int}@Q \rightarrow \text{Int}@P \times \text{Int}@Q$$

and represents the shared key as a pair of equal keys, one for each participant.

The choreography then takes a shared key as its parameter and produces a pair of unidirectional channels that wrap the communication primitive with the necessary encryption based on the key:

```
makeSecureChannels(P, Q) = λkey : Int@P × Int@Q.
  Pair (λval : String@P. (dec(Q) (snd key) (comP,Q (enc(P) (fst key) val))))
    (λval : String@Q. (dec(P) (fst key) (comQ,P (enc(Q) (snd key) val))))
```

Here `enc` and `dec` are local function for encoding and decoding values based on keys.

The fact that this choreography returns a pair of channels can also be seen from its type:

$$(\text{Int}@P \times \text{Int}@Q) \rightarrow ((\text{String}@P \rightarrow \text{String}@Q) \times (\text{String}@Q \rightarrow \text{String}@P))$$

Using the channels is as easy as using `com` itself and amounts to a function application.

Process language

To implement recursive functions in Chor λ , we also add recursive functions to our process language: $B ::= \dots \mid f(\vec{p})$. They have the same syntax as in choreographies, being parametric on the names of any other processes our process may interact with as part of the function. Local function names are associated with their definition by a function \mathbb{D} , which works the same as D in the choreographic setting. Furthermore, we add a transition rule to the process language similar to rule DEF for choreographies.

Endpoint Projection

We respectively project function calls, type variables, and function definitions as follows.

$$\begin{aligned} \llbracket f(\vec{p}) \rrbracket_{\mathbf{p}} &= \begin{cases} f_i(\mathbf{p}_1, \dots, \mathbf{p}_{i-1}, \mathbf{p}_{i+1}, \dots, \mathbf{p}_n) & \text{if } \vec{p} = \mathbf{p}_1, \dots, \mathbf{p}_{i-1}, \mathbf{p}, \mathbf{p}_{i+1}, \dots, \mathbf{p}_n \\ \perp & \text{otherwise} \end{cases} \\ \llbracket t@{\vec{p}} \rrbracket_{\mathbf{p}} &= \begin{cases} t_i & \text{if } \vec{p} = \mathbf{p}_1, \dots, \mathbf{p}_{i-1}, \mathbf{p}, \mathbf{p}_{i+1}, \dots, \mathbf{p}_n \\ \perp & \text{otherwise} \end{cases} \\ \llbracket D \rrbracket &= \{f_i(\mathbf{p}_1, \dots, \mathbf{p}_{i-1}, \mathbf{p}_{i+1}, \dots, \mathbf{p}_n) \mapsto \llbracket M \rrbracket_{\mathbf{p}_i} \mid D(f(\mathbf{p}_1, \dots, \mathbf{p}_n)) = M\} \end{aligned}$$

Each named function gets projected to a different named function for each process in its list of parameters, with the projected environment now treating each of these as separate functions parametric on the remaining involved processes. These parameters are needed to implement interactions. Each process can enter a named function independently. Thus, for example, if $D(f(\mathbf{p}, \mathbf{q})) = M$ we get $\llbracket D \rrbracket (f_1(\mathbf{q})) = \llbracket M \rrbracket_{\mathbf{p}}$ and $\llbracket D \rrbracket (f_2(\mathbf{p})) = \llbracket M \rrbracket_{\mathbf{q}}$.

On the other hand, projection of recursive types does not need to consider other processes than the one we are projecting on, since local types never mention any processes. Σ is otherwise projected similarly to D . For example, if $t@(\mathbf{p}, \mathbf{q}) = T \in \Sigma$ then $t_1 = \llbracket T \rrbracket_{\mathbf{p}} \in \llbracket \Sigma \rrbracket$ and $t_2 = \llbracket T \rrbracket_{\mathbf{q}} \in \llbracket \Sigma \rrbracket$.

$$\llbracket \Sigma \rrbracket = \{t_i = \llbracket T \rrbracket_{\mathbf{p}_i} \mid t@(\mathbf{p}_1, \dots, \mathbf{p}_n) = T \in \Sigma\}$$

► **Example 23** (Projecting Example 21). Projecting the choreography in Example 21 yields the processes `remoteMap1` (for the client) and `remoteMap2` (for the server) below. The bodies of `remoteFunction1` and `remoteFunction2` are the terms in Example 11.

```
remoteMap1(S) = λfun : ⊥. λlist : [Int].
  case list of
    Inl x ⇒ ⊕S stop ();
    Inr x ⇒ ⊕S again
      cons1 (remoteFunction1(S) ⊥ (fst x)) (remoteMap1(S) ⊥ (snd x))
remoteMap2(C) = λfun : Int → Int. λlist : ⊥.
  &C{stop : ⊥, again : (remoteFunction2(C) fun ⊥) (remoteMap2(C) fun ⊥)}
```

► **Example 24** (Projecting Example 22). Projecting our choreographies `diffieHellman(P, Q)` and `makeSecureChannels(P, Q)` for process P yields the following behaviours.

```
\llbracket D(diffieHellman(P, Q)) \rrbracket1 (Q) = λpsk : Int. λqsk : ⊥. λpsg : Int. λqsg : ⊥. λpsp : Int. λqsp : ⊥.
  pair (modPow1 psg (recvQ ⊥)) psp
    (sendQ (modPow1 psg psk psp))
```

$$\begin{aligned} \llbracket D(\text{makeSecureChannels}(P, Q)) \rrbracket_1(Q) &= \lambda key : \text{Int} \times \perp. \\ \mathbf{Pair} (\lambda val : \text{String}. ((\mathbf{snd} \ key) (\mathbf{send}_Q (\mathbf{encrypt}_1 (\mathbf{fst} \ key) \ val)))) \\ (\lambda val : \perp. (\mathbf{decrypt}_1 (\mathbf{fst} \ key) (\mathbf{recv}_Q (\mathbf{snd} \ key)))) \end{aligned}$$

Note the way function calls such as $\text{modPow}(P)$ in the choreography get projected to modPow_1 on P , since they are treated as degenerate choreographies (they have only one process) and P is the first and only process involved. Conversely, $\text{modPow}(Q)$ on P gets projected as \perp since it is located entirely at a different process.

4.2 Out-of-order execution

In the presence of recursion, getting a correspondence between a process and choreographic language becomes much more challenging. In our results for $\text{Chor}\lambda$ without recursion, we relied on the fact that a choreography would eventually reduce to a value. This is no longer true as choreographies can now diverge, and worse they can diverge at one process without diverging at another. Let, for example, $M = (\lambda x : \text{Int}@p. \mathbf{fst} (\mathbf{Pair} \ 5@q \ x)) \ f(p)$. Assume that $D(f(p_1)) = M'$, where M' diverges. Then the reduction rules that we have seen so far would not allow x to be instantiated. However, $\llbracket f(p) \rrbracket_q = \perp$, so $\llbracket M \rrbracket_q$ can reduce to 5 . Therefore, we need a way to let M copy the reduction of $\mathbf{fst} (\mathbf{Pair} \ 5@q \ x)$ to $5@q$. In [6], we included corresponding reduction rules for $\text{Chor}\lambda$ to deal with this kind of issues. These rules are all type preserving and avoid creating situations where processes disagree on which communication should be performed first [6]. These rules were unnecessary to deal with the recursion-free fragment, so we introduce them now.

Rule INABS below addresses situations as in the previous example.

$$\frac{M \xrightarrow{\ell, P}_D M'}{\lambda x : T.M \xrightarrow{\lambda, P}_D \lambda x : T.M'} \text{ [INABS]} \quad \frac{M \xrightarrow{\ell, R}_D M' \quad \ell = \lambda \Rightarrow P \cap \text{pn}(N) = \emptyset}{M \ N \xrightarrow{\tau, P}_D M' \ N} \text{ [APP1]}$$

Rule APP1 use the ℓ -component in reduction labels to identify whether a reduction is performed under an abstraction ($\ell = \lambda$) or not ($\ell = \tau$). We need this distinction to prevent interactions under an abstraction performed by processes involved in the righthandside of an application. This restriction serves to avoid breaking causal dependencies between communications. Consider the choreography $(\lambda x : \text{Int}@p. \mathbf{com}_{q,p} \ 4@q) (\mathbf{com}_{q,p} \ 5@q)$, where the righthandside communication should be performed first – without the restriction, this would not be guaranteed. Reductions under abstractions additionally necessitates a new safety condition on rule APPABS , ensuring that the free variables of V are distinct from the bound variables of M to avoid problems with scope.

Our modification allows the choreography $M = (\lambda x : \text{Int}@q. \mathbf{fst} (\mathbf{Pair} \ 5@q \ x)) \ f(p, q)$ to reduce to $M' = (\lambda x : \text{Int}@p_2. 5@q) \ f(p, q)$. Thus, the projections of M on p and q must be able to reduce to the projections of M' . For p this is easy, since $\llbracket M \rrbracket_p = \llbracket M' \rrbracket_p = \perp \ f_1(q)$. For q , however, we need $\llbracket M \rrbracket_q = (\lambda x : \text{Int}. \mathbf{fst} (\mathbf{Pair} \ \perp \ x)) \ f_2(p)$ to reduce to $\llbracket M' \rrbracket_q = (\lambda x : \text{Int}. \perp) \ f_2(p)$, which requires the process language to have similar out-of-order semantics. We therefore add an equivalent rule NINABS and modify rule NAPP1 similarly to rule APP1 .

In the network, rather than checking for interacting processes, we do not allow communication actions (\mathbf{send} , \mathbf{recv} , \oplus , $\&$) from inside an abstraction. The reduction labels for the process language are thus simpler (τ or λ), since we do not need to track process names involved in actions.

Similar problems appear with applications that have divergent subterms on the lefthand-side, like $f(q) ((\lambda x : \text{Int}@p. 4@q) \ 3@p)$, and are treated similarly (the corresponding reduction rules are given in the appendix).

$$\begin{array}{c}
\frac{x \notin \text{fv}(M')}{((\lambda x : T.M) N) M' \rightsquigarrow (\lambda x : T.(M M')) N} \text{ [R-ABS R]} \\
\frac{x, x' \notin \text{fv}(M) \quad \text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\text{case } N \text{ of } \text{Inl } x \Rightarrow M_1; \text{Inr } x' \Rightarrow M_2) \rightsquigarrow \text{case } N \text{ of } \text{Inl } x \Rightarrow (M M_1); \text{Inr } x' \Rightarrow (M M_2)} \text{ [R-CASE L]} \\
\frac{\text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\text{select}_{q,p} l N) \rightsquigarrow \text{select}_{q,p} l (M N)} \text{ [R-SELL]} \\
\frac{y \text{ fresh for } M}{\lambda x : T.M \rightsquigarrow \lambda y : T.M[x := y]} \text{ [R-ALPH]}
\end{array}$$

■ **Figure 5** Rewriting of Chor λ (representative rules).

$$\begin{array}{c}
\frac{\text{pn}(B) = \emptyset}{B (\&_p \{l_1 : B_1, \dots, l_n : B_n\}) \rightsquigarrow \&_p \{l_1 : B B_1, \dots, l_n : B B_n\}} \text{ [LR-OFFL]} \\
\frac{\text{pn}(B') = \emptyset}{B' (\oplus_p l B) \rightsquigarrow \oplus_p l (B' B)} \text{ [LR-CHOL]} \quad \frac{}{\perp \perp \rightsquigarrow \perp} \text{ [LR-BOTM]}
\end{array}$$

■ **Figure 6** Rewriting of behaviours (representative rules).

Dealing with recursive functions in nested applications requires another addition to the semantics of Chor λ . Consider the choreography $M = ((\lambda x : \text{Int}@p.\lambda y : \text{Int}@q.3@p) f(p)) 4@q$. We have $\llbracket M \rrbracket_q = ((\lambda x : \perp.\lambda y : \text{Int}.\perp) \perp) 4$, which can reduce to \perp in two steps. Reducing M accordingly requires being able to instantiate y as $4@q$ even if $f(p)$ diverges. For this, and other cases of functions whose divergence blocks actions, Chor λ has a set of rewriting rules (see Figure 5). In our example, M can be rewritten as $(\lambda x : \text{Int}@p.(\lambda y : \text{Int}@q.3@p 4@q)) f(p)$ by using rule R-ABS R, which can reduce to $(\lambda x : \text{Int}@p.3@p) f(p)$ as needed. In the rewriting rules that move a subterm in a lefthandside further in, the synchronising processes of the subterm, $\text{spn}(M)$, is used to prevent rewritings that would change the order of communications. To use the rewritings in the semantics we add the rule

$$\frac{M \rightsquigarrow^* N \quad N \xrightarrow{\tau, P} M'}{M \xrightarrow{\tau, P}_D M'} \text{ [STR]}$$

As before, equivalent rules must be added to the semantics of our process language (see Figure 6), and the reduction relation is closed under these rewritings. This allows $\llbracket M \rrbracket_p = ((\lambda x : \text{Int}.\lambda y : \perp.3) f_1()) \perp$ to be rewritten to $(\lambda x : \text{Int}.\lambda y : \perp.3 \perp) f_1()$, which can reduce to $(\lambda x : \text{Int}.3) f_1()$.

4.3 Properties

Thanks to the extensions discussed in this section, our results can be generalised to the full language of Chor λ with recursion.

► **Theorem 25 (Completeness).** *Given a closed choreography M , if $M \xrightarrow{\tau, P}_D M'$ and $\Theta; \Sigma; \Gamma \vdash M : T$ and $\llbracket M \rrbracket$ is defined, then there exist networks \mathcal{N} and M'' such that: $\llbracket M \rrbracket \rightarrow_{\llbracket D \rrbracket}^+ \mathcal{N}$; $M' \rightarrow^* M''$; and $\mathcal{N} \cong \llbracket M'' \rrbracket$.*

► **Theorem 26** (Soundness). *Given a closed choreography M , if $\Theta; \Gamma \vdash M : T$ and $\llbracket M \rrbracket \rightarrow^* \mathcal{N}$ for some network \mathcal{N} , then there exist a choreography M' , and a network \mathcal{N}' such that: $M \rightarrow_D^* M'$; $\mathcal{N} \rightarrow^* \mathcal{N}'$; and $\mathcal{N}' \cong \llbracket M' \rrbracket$.*

From Theorems 25 and 26 and the type preservation and progress results from [6], we get the following corollary about deadlock-freedom. Specifically, the EPP of a well-typed closed choreography can keep reducing until all processes contain only local values (which denotes termination).

► **Corollary 27** (Deadlock-freedom). *Given a closed choreography M and a function environment D containing all the functions of M , if $\Theta; \Sigma; \Gamma \vdash M : T$ and $\Theta; \Sigma; \Gamma \vdash D$, then: whenever $\llbracket M \rrbracket \rightarrow_{[D]}^* \mathcal{N}$ for some network \mathcal{N} , either there exists P and \mathcal{N}' such that $\mathcal{N} \xrightarrow{Tp}_{[D]} \mathcal{N}'$ or $\mathcal{N} = \prod_{p \in \text{pn}(M)} p[L_p]$.*

We also show that adding recursion does not stop our projection being modular.

► **Proposition 28.** *The EPP $\llbracket - \rrbracket$ given in Definition 8 and extended with the equations in Section 4.1 is modular.*

Proof. The only change to the projection of choreographies is adding the projection of $f(\vec{p})$, for which Lemma 13 still holds. Since no new contexts have been added, projection is then still modular. ◀

5 EAP

We now use our theory of EPP to obtain an implementation of the core of the Extensible Authentication Protocol (EAP) [28], which was modelled as a choreography in [6]. EAP is a widely-employed link-layer protocol that defines an authentication framework allowing a peer P to authenticate with a backend authentication server S , with the communication passing through an authenticator A that acts as an access point for the network.

The framework provides a core protocol parametrised over a set of authentication methods (either predefined or custom vendor-specific ones), modelled as individual choreographies with type $\text{AuthMethod}@(\mathcal{P}, \mathcal{A}, \mathcal{S}) = \text{String}@S \rightarrow_{\{\mathcal{P}, \mathcal{A}\}} \text{Bool}@S$.

For reasons of modularity, it is desirable that the core of the protocol be written in a way that does not assume any particular authentication method. The $\text{eap}(\mathcal{P}, \mathcal{A}, \mathcal{S})$ choreography does exactly that by leveraging higher-order composition of choreographies:

```
eap(P, A, S) = λmethods : [AuthMethod]@(P, A, S).
  eapAuth(P, A, S) (eapIdentity(P, A, S) "Auth request"@S) methods

eapAuth(P, A, S) = λid : String@S. λmethods : [AuthMethod]@(P, A, S).
  if empty(P, A, S) methods then
    eapFailure(P, A, S) "Try again later"@S
  else
    if (fst methods) id then
      selectS,P ok (selectS,A ok (eapSuccess(P, A, S) "Welcome"@S))
    else
      selectS,P ko (selectS,A ko (eapAuth(P, A, S) id (snd methods)))
```

For the sake of simplicity, we have left out the definitions of a couple of helper choreographies that are referenced in the example:

```
eapIdentity(P, A, S) : String@S →\{P,A\}} String@S
```

$$\begin{aligned} \text{empty}(P, A, S) &: [\text{AuthMethod}]@(\text{P}, \text{A}, \text{S}) \rightarrow \text{Bool}@(\text{P}, \text{A}, \text{S}) \\ \text{eapSuccess}(P, A, S) &: \text{String}@S \rightarrow (\text{String}@P \times \text{String}@A) \\ \text{eapFailure}(P, A, S) &: \text{String}@S \rightarrow (\text{String}@P \times \text{String}@A) \end{aligned}$$

First, $\text{eap}(P, A, S)$ fetches the client's identity using $\text{eapIdentity}(P, A, S)$, a function which exchanges the necessary EAP packets and delivers the client's identity to the server. Once the identity is known, $\text{eapAuth}(P, A, S)$ is invoked in order to try the list of authentication methods until one succeeds, or the list is exhausted and authentication fails.

EAP is parametric on a list of choreographies called *methods*. We use the notation for lists in $[\text{AuthMethod}]@(\text{P}, \text{A}, \text{S})$ as described in Example 21, as well as the **if** M **then** M' **else** M'' construct which is just syntactic sugar for the previously described **case** M **of** $\text{Inl } x \Rightarrow M'$; $\text{Inr } x \Rightarrow M''$. Each authentication method can be an arbitrarily-complex choreography with its own communication structures that can involve all three involved processes, and it implements a particular authentication method on top of EAP.

The function $\text{empty}(P, A, S)$ is used to determine whether the list of methods is empty. Recall the distributed boolean from Example 15, and note how we now use the same idea to minimise unnecessary communication while still guaranteeing that every process has the necessary information. The return type of this function, $\text{Bool}@(\text{P}, \text{A}, \text{S})$, denotes that the function uniformly returns either true ($\text{Inl } ()$) or false ($\text{Inr } ()$) at all of P , A , and S . That is, the result is guaranteed to be the same at these three processes. Since agreement is guaranteed, each process can locally check its own value without having to perform any selections. This is in contrast to the return type of each authentication method, $\text{Bool}@S$, meaning that only the server S has the authority of determining whether the authentication method was successful or not.

Finally, depending on the outcome of the authentication, an appropriate EAP packet is delivered by using either $\text{eapSuccess}(P, A, S)$ or $\text{eapFailure}(P, A, S)$ to indicate the result to the client.

```
eap1(A, S) = λmethods : [AuthMethod].
  eapAuth1(A, S) (eapIdentity1(A, S) ⊥) methods

eap2(P, S) = λmethods : [AuthMethod].
  eapAuth2(P, S) (eapIdentity2(P, S) ⊥) methods

eap3(P, A) = λmethods : [AuthMethod].
  eapAuth3(P, A) (eapIdentity3(P, A) "Auth request") methods
```

It is interesting to look at the projections of $\text{eapAuth}(P, A, S)$ for each of the three participants, which follow below. For the purposes of projection, we desugar the if-then-else construct.

```
eapAuth1(A, S) = λid : ⊥. λmethods : [AuthMethod].
  case empty1(A, S) methods of
    Inl _ ⇒ eapFailure1(A, S) ⊥
    Inr _ ⇒ &S {ok : eapSuccess1(A, S) ⊥
              ko : eapAuth1(A, S) ⊥ (snd methods)}

eapAuth2(P, S) = λid : ⊥. λmethods : [AuthMethod].
  case empty2(P, S) methods of
    Inl _ ⇒ eapFailure2(P, S) ⊥
    Inr _ ⇒ &S {ok : (eapSuccess2(P, S) ⊥)
              ko : (eapAuth2(P, S) ⊥ (snd methods))}
```

```

eapAuth3(P, A) = λid : String. λmethods : [AuthMethod].
  case empty3(P, A) methods of
    Inl _ ⇒ eapFailure3(P, A) "Try again later"
    Inr _ ⇒ case (fst methods) id of
      Inl _ ⇒ ⊕P ok (⊕A ok (eapSuccess3(P, A) "Welcome"))
      Inr _ ⇒ ⊕P ko (⊕A ko (eapAuth3(P, A) id (snd methods)))

```

Note that the implementation of the check `empty(P, A, S) methods` at each process is completely local, i.e., it does not perform communications. This is possible because all processes have access to the same list. Afterwards however, only the server `S` is capable of determining whether the authentication method was successful or not, and has to communicate that result to the other two participants by means of selections.

6 Related Work

We already discussed the most related work on choreographic programming and EPP in Section 1. In this section, we discuss some technical aspects of our development in the context of previous work more in detail.

In our process language, the terms for communication actions (send, receive, selection, and branching) are adaptations to the functional setting of standard primitives from traditional imperative choreographic programming [8, 10, 21] and the local language of multiparty session types (choreographies without computation) [20, 19, 3]. A similar adaptation was carried out in [27] for the different setting of multi-threading (their primitives are not based on process names, but shared channels). Modelling a network as a map from process names to programs was previously done in [9, 24]. The idea of reporting the names of the involved processes in transition labels comes from [2, 19, 9, 24].

The first attempt at adding higher-order composition to choreographies goes back to [11], for a choreographic language that cannot express data nor computation (it is an abstract specification language). The approach in [11] adopts centralised coordination: resolving a choreographic application ($M M'$ in $\text{Chor}\lambda$, with M' involving more than one process) requires that the programmer picks a process as central coordinator, which then orchestrates the other processes with multicasts. This coordination effectively acts as a barrier, so processes cannot perform their own local computations independently of each other when higher-order composition is involved. Ten years after [11], another attempt at a notion of EPP for higher-order choreographies was proposed in [18]. The language in [18] is more expressive, i.e., it supports expressing computation at processes. However, this feature came at a cost: it is even more centralised than [11]. In particular, every application in a choreography requires that all processes generated by projection go through a global barrier that involves the entire system. The global barrier is modelled as a middleware in the semantics of the language, and involves even processes that do not contribute at all to the function or its arguments. Because processes need to participate also in the resolution of applications that do not involve them, the notion of EPP in [18] is not modular.

In contrast to [11] and [18], $\text{Chor}\lambda$ presents no “hidden” barriers: coordination among processes is left to the programmer of the choreography, and EPP inserts no hidden synchronisations. Our EPP thus generates more concurrent and faithful implementations. It is also the first modular EPP for functional choreographic programming: changing the behaviours of some processes in a choreography requires re-running EPP only for those processes. This is important for the application of choreographic programming to DevOps (continuous integration and deployment), library management, and modularity in general.

Another related line of work is that on multitier programming and its progenitor calculus, Lambda 5 [25]. Similarly to Chor λ , Lambda 5 and multitier languages have data types with locations [29]. However, they are used very differently. In choreographic languages (thus Chor λ), programs have a “global” point of view and express how multiple processes interact with each other. By contrast, in multitier programming programs have the usual “local” point of view of a single process but they can nest (local) code that is supposed to be executed remotely. The reader interested in a detailed comparison of choreographic and multitier programming can consult [17], which presents algorithms for translating choreographies to multitier programs and vice versa. The correctness of these algorithms has never been proven, because they use an informally-specified fragment of Choral as a representative choreographic language. We conjecture that the introduction of an EPP for Chor λ could be the basis for a future comparison of the compilations for choreographic programs (in terms of Chor λ) and multitier programs (in terms of Lambda 5).

To the best of our knowledge, no other work supports distributed choice types. The nearest feature is presented in [21], where choreographic conditionals for a first-order calculus can be conjunctions of local conditions at different processes. These conditions must be checked to be consistent by means of separate proofs given in a Hoare-like logic. Our syntax is more general, since conditions can be choreographies, and our EPP requires no such additional proofs. However, using a Hoare logic in [21] gives some interesting flexibility, in that agreement does not need to be encoded as distributed sum types. In the future, it could be interesting to integrate the two approaches such that agreement could be proved by using a logic and then made manifest to EPP through our distributed choice types.

7 Conclusion and Future Work

We have presented a new theory of compilation for higher-order functional choreographies, which introduces modularity and decentralisation.

Our development validates the design of Chor λ [6], but it also reveals that in the case without recursion it can be significantly simplified: reduction rules for out-of-order execution were not necessary until we had to deal with divergence. In particular, we have shown that the fragment of Chor λ without recursion can be modelled by simple semantics and still achieve the standard deadlock-freedom by design property. However, once recursion is added, a more sophisticated semantics allowing for out-of-order execution is required. This stems from the structure of a functional choreography being different than traditional imperative choreographies.

Our study fills a knowledge gap that is relevant for the future implementations and applications of choreographic languages. An ad-hoc distributed implementation of higher-order choreographies exists already in the Choral programming language [16]. However, Choral is a large object-oriented language that extends Java, meaning that it is not practical to formally study and prove the standard results expected of a choreographic language. We have been able to prove these results – correspondence between choreography and projected distributed implementation (Theorems 25 and 26) and deadlock-freedom (Corollary 27) – because Chor λ captures the essence of higher-order choreographic composition in a small language based on the λ -calculus. Our EPP is largely consistent with the implementation of the Choral compiler, but there are two key differences, both caused by Chor λ being based on the λ -calculus. First, since Choral is an object-oriented language, not every expression needs to return a value even if the result of the expression is located elsewhere as in **send**; therefore, Choral does not need a \perp construct. Second, Choral does not have distributed choice types and instead restricts all conditions to be local (at one process). Thus, our distributed choice types could form the basis for an interesting extension of Choral.

Aside from Choral, existing choreographic programming languages either have no higher-order constructs (e.g., Scribble [30], a language based on multiparty session types [19]), or have the compilation of their higher-order constructs lack modularity and decentralisation (e.g., Pirouette [18]). Our results provide a foundation for adding mechanisms for higher-order composition to other choreographic and similar languages with modular compilation.

Future Work

Synchronous communication is widely adopted in theories of processes and is usually implemented in practice by using acknowledgements. A potential extension of $\text{Chor}\lambda$ is adding support for asynchronous communication, which is usually achieved by adding message queues and choreographic terms to represent partially-executed communications [12, 7, 14, 24].

Another potential extension of $\text{Chor}\lambda$, our process language, and our theory of EPP would be to enable abstraction over process names, that is, extending the syntax such that values can be the names of processes to be acted upon. This could, for example, enable the modelling of choreographies with dynamic topologies, where processes discover whom they have to interact with at runtime.

References

- 1 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.
- 2 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.
- 3 Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54(3):243–269, 2017. doi:10.1007/s00236-016-0285-y.
- 4 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:24)2012.
- 5 Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932. URL: <http://www.jstor.org/stable/1968337>.
- 6 Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. Functional choreographic programming. In Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu, editors, *Theoretical Aspects of Computing – ICTAC 2022 – 19th International Colloquium, Tbilisi, Georgia, September 27-29, 2022, Proceedings*, volume 13572 of *Lecture Notes in Computer Science*, pages 212–237. Springer, 2022. doi:10.1007/978-3-031-17715-6_15.
- 7 Luís Cruz-Filipe and Fabrizio Montesi. On asynchrony and choreographies. In Massimo Bartoletti, Laura Bocchi, Ludovic Henrio, and Sophia Knight, editors, *Proceedings 10th Interaction and Concurrency Experience, ICE@DisCoTec 2017, Neuchâtel, Switzerland, 21-22nd June 2017*, volume 261 of *EPTCS*, pages 76–90, 2017. doi:10.4204/EPTCS.261.8.
- 8 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.
- 9 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021 – 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.
- 10 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a turing-complete choreographic language in coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPICs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ITP.2021.15.

- 11 Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 – Concurrency Theory – 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi:10.1007/978-3-642-32940-1_20.
- 12 Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming – 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013. doi:10.1007/978-3-642-39212-2_18.
- 13 Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976. doi:10.1109/TIT.1976.1055638.
- 14 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPICs*, pages 11:1–11:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ECOOP.2017.11.
- 15 Saverio Giallorenzo, Ivan Lanese, and Daniel Russo. Chip: A choreographic integration process. In Hervé Panetto, Christophe Debruyne, Henderik A. Proper, Claudio Agostino Ardagna, Dumitru Roman, and Robert Meersman, editors, *Procs. OTM, part II*, volume 11230 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2018. doi:10.1007/978-3-030-02671-4_2.
- 16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Object-oriented choreographic programming. *CoRR*, abs/2005.09520, 2020. URL: <https://arxiv.org/abs/2005.09520>.
- 17 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. Multiparty Languages: The Choreographic and Multitier Cases. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 12-17, 2021, Aarhus, Denmark (Virtual Conference)*, *LIPICs*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2021. To appear. Pre-print available at <https://fabriziomontesi.com/files/gmprsw21.pdf>.
- 18 Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:10.1145/3498684.
- 19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: *POPL*, pages 273–284, 2008. doi:10.1145/2827695.
- 20 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.
- 21 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies – computing preconditions in choreographic programming. In Ilya Sergey, editor, *Programming Languages and Systems – 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 520–547. Springer, 2022. doi:10.1007/978-3-030-99336-8_19.
- 22 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proc. of ASPLOS*, pages 517–530, 2016.
- 23 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. <http://www.fabriziomontesi.com/files/choreographic-programming.pdf>.
- 24 Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.
- 25 Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 286–295. IEEE Computer Society, 2004. doi:10.1109/LICS.2004.1319623.

- 26 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978. doi:10.1145/359657.359659.
- 27 Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multi-threaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006. doi:10.1016/j.tcs.2006.06.028.
- 28 John Vollbrecht, James D. Carlson, Larry Blunk, Dr. Bernard D. Aboba, and Henrik Levkowetz. Extensible Authentication Protocol (EAP). RFC 3748, June 2004. doi:10.17487/RFC3748.
- 29 Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Comput. Surv.*, 53(4):81:1–81:35, 2020. doi:10.1145/3397495.
- 30 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing – 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

A Full definitions and proofs

► **Definition 29** (Free Variables). *Given a choreography M , the free variables of M , $\text{fv}(M)$ are defined as:*

$$\begin{aligned}
\text{fv}(N \ N') &= \text{fv}(N) \cup \text{fv}(N') & \text{fv}(\mathbf{select}_{q,p} \ l \ M) &= \text{fv}(M) \\
\text{fv}(x) &= x & \text{fv}(\lambda x : T.N) &= \text{fv}(N) \setminus \{x\} \\
\text{fv}(()@p) &= \emptyset & \text{fv}(\mathbf{com}_{q,p}) &= \emptyset \\
\text{fv}(f(\vec{p})) &= \emptyset & \text{fv}(\mathbf{Pair} \ V \ V') &= \text{fv}(V) \cup \text{fv}(V') \\
\text{fv}(\mathbf{case} \ N \ \mathbf{of} \ \mathbf{Inl} \ x \Rightarrow M; \ \mathbf{Inr} \ y \Rightarrow M') &= \text{fv}(N) \cup (\text{fv}(M) \setminus \{x\}) \cup (\text{fv}(M') \setminus \{y\}) \\
\text{fv}(\mathbf{fst}) &= \text{fv}(\mathbf{snd}) = \emptyset & \text{fv}(\mathbf{Inl} \ V) &= \text{fv}(\mathbf{Inr} \ V) = \text{fv}(V)
\end{aligned}$$

► **Definition 30** (Bound Variables). *Given a choreography M , the bound variables of M , $\text{bv}(M)$ are defined as:*

$$\begin{aligned}
\text{bv}(N \ N') &= \text{bv}(N) \cup \text{bv}(N') & \text{bv}(\mathbf{select}_{q,p} \ l \ M) &= \text{bv}(M) \\
\text{bv}(x) &= \emptyset & \text{bv}(\lambda x : T.N) &= \text{bv}(N) \cup \{x\} \\
\text{bv}(()@p) &= \emptyset & \text{bv}(\mathbf{com}_{q,p}) &= \emptyset \\
\text{bv}(f(\vec{p})) &= \emptyset & \text{bv}(\mathbf{Pair} \ V \ V') &= \text{bv}(V) \cup \text{bv}(V') \\
\text{bv}(\mathbf{case} \ N \ \mathbf{of} \ \mathbf{Inl} \ x \Rightarrow M; \ \mathbf{Inr} \ y \Rightarrow M') &= \text{bv}(N) \cup \text{bv}(M) \cup \{x\} \cup (\text{bv}(M') \cup \{y\}) \\
\text{bv}(\mathbf{fst}) &= \text{bv}(\mathbf{snd}) = \emptyset & \text{bv}(\mathbf{Inl} \ V) &= \text{bv}(\mathbf{Inr} \ V) = \text{bv}(V)
\end{aligned}$$

► **Definition 31** (Process names of a type). *The process names of a type T , $\text{pn}(T)$, are defined as follows.*

$$\begin{aligned}
\text{pn}(t@R) &= \vec{R} & \text{pn}(T \rightarrow_{\rho} T') &= \text{pn}(T) \cup \text{pn}(T') \cup \rho \\
\text{pn}(()@R) &= \{R\} & \text{pn}(T + T') &= \text{pn}(T \times T') = \text{pn}(T) \cup \text{pn}(T')
\end{aligned}$$

► **Definition 32** (Process names of a choreography). *The process names of a choreography M , $\text{pn}(M)$, are defined as follows.*

$$\begin{aligned}
\text{pn}(M \ N) &= \text{pn}(M) \cup \text{pn}(N) \\
\text{pn}(\mathbf{select}_{p,q} \ l \ M) &= \{p, q\} \cup \text{pn}(M) \\
\text{pn}(x) &= \emptyset \\
\text{pn}(\mathbf{case} \ M \ \mathbf{of} \ \mathbf{Inl} \ x \Rightarrow N; \ \mathbf{Inr} \ y \Rightarrow N') &= \text{pn}(M) \cup \text{pn}(N) \cup \text{pn}(N') \\
\text{pn}(\lambda x : T.M) &= \text{pn}(T) \cup \text{pn}(M) \\
\text{pn}(\mathbf{Inl} \ V) &= (\text{pn} \ \mathbf{Inr} \ V) = \text{pn}(V)
\end{aligned}$$

$$\begin{array}{c}
\frac{\Theta'; \Sigma; \Gamma, x : T \vdash M : T' \quad \rho \cup \text{pn}(T) \cup \text{pn}(T') = \Theta' \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \lambda x : T. M : T \rightarrow_{\rho} T'} \text{[TABS]} \\
\frac{x : T \in \Gamma \quad \text{pn}(T) \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash x : T} \text{[TVAR]} \quad \frac{\Theta; \Sigma; \Gamma \vdash N : T \rightarrow_{\rho} T' \quad \Theta; \Sigma; \Gamma \vdash M : T}{\Theta; \Sigma; \Gamma \vdash N M : T'} \text{[TAPP]} \\
\frac{\Theta; \Sigma; \Gamma \vdash N : T_1 + T_2 \quad \Theta; \Sigma; \Gamma, x : T_1 \vdash M' : T \quad \Theta; \Sigma; \Gamma, x' : T_2 \vdash M'' : T}{\Theta; \Sigma; \Gamma \vdash \mathbf{case} N \mathbf{ of} \mathbf{Inl} x \Rightarrow M'; \mathbf{Inr} x' \Rightarrow M'' : T} \text{[TCASE]} \\
\frac{\Theta; \Sigma; \Gamma \vdash M : T \quad \mathbf{q}, \mathbf{p} \in \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{select}_{\mathbf{q}, \mathbf{p}} l M : T} \text{[TSEL]} \\
\frac{f(\vec{p}) : T \in \Gamma \quad \text{pn}(T) \subseteq \vec{p} \subseteq \Theta \quad \|\vec{p}\| = \|\vec{p}'\| \quad \text{distinct}(\vec{p})}{\Theta; \Sigma; \Gamma \vdash f(\vec{p}) : T[\vec{p}' := \vec{p}]} \text{[TFUN]} \\
\frac{\mathbf{p} \in \Theta}{\Theta; \Sigma; \Gamma \vdash ()@_{\mathbf{p}} : ()@_{\mathbf{p}}} \text{[TUNIT]} \quad \frac{\mathbf{q}, \mathbf{p} \in \Theta \quad \text{pn}(T) = \mathbf{q}}{\Theta; \Sigma; \Gamma \vdash \mathbf{com}_{\mathbf{q}, \mathbf{p}} : T \rightarrow_{\emptyset} T[\mathbf{q} := \mathbf{p}]} \text{[TCOM]} \\
\frac{\Theta; \Sigma; \Gamma \vdash V : T \quad \Theta; \Sigma; \Gamma \vdash V' : T'}{\Theta; \Sigma; \Gamma \vdash \mathbf{Pair} V V' : (T \times T')} \text{[TPAIR]} \\
\frac{\text{pn}(T \times T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{fst} : (T \times T') \rightarrow_{\emptyset} T} \text{[TPROJ1]} \quad \frac{\text{pn}(T \times T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{snd} : (T \times T') \rightarrow_{\emptyset} T'} \text{[TPROJ2]} \\
\frac{\Theta; \Sigma; \Gamma \vdash V : T \quad \text{pn}(T + T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{Inl} V : (T + T')} \text{[TINL]} \quad \frac{\Theta; \Sigma; \Gamma \vdash V : T' \quad \text{pn}(T + T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{Inr} V : (T + T')} \text{[TINR]} \\
\frac{\Theta; \Sigma; \Gamma \vdash M : t@_{\vec{p}} \quad t@_{\vec{p}'} =_{\Sigma} T \quad \|\vec{p}\| = \|\vec{p}'\| \quad \text{distinct}(\vec{p})}{\Theta; \Sigma; \Gamma \vdash M : T[\vec{p}' := \vec{p}]} \text{[TEQ]} \\
\frac{\forall f(\vec{p}) \in \text{dom}(D) : f(\vec{p}) : T \in \Gamma \quad \vec{p}; \Sigma; \Gamma \vdash D(f(\vec{p})) : T \quad \text{distinct}(\vec{p}) \quad \vec{p} \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash D} \text{[TDEFS]}
\end{array}$$

■ **Figure 7** Full set of typing rules for Chor λ .

$$\begin{aligned}
\text{pn}(\mathbf{Pair} V V') &= \text{pn}(V) \cup \text{pn}(V') \\
\text{pn}(\mathbf{fst}) &= \text{pn}(\mathbf{snd}) = \emptyset \\
\text{pn}(\mathbf{com}_{\mathbf{p}, \mathbf{q}}) &= \{\mathbf{p}, \mathbf{q}\}
\end{aligned}$$

► **Definition 33.** We define the set of synchronising processes of a choreography M , $\text{spn}(M)$, by recursion on the structure of M :

$$\begin{aligned}
\text{spn}(\mathbf{com}_{S,R}) &= \{S, R\}, \text{spn}(\mathbf{select}_{S,R} l M) = \{S, R\} \cup \text{spn}(M), \\
\text{spn}(f(\vec{R})) &= \vec{R}, \text{ and homomorphically on all other cases.}
\end{aligned}$$

► **Definition 34** (Merging). Given two behaviours B and B' , $B \sqcup B'$ is defined as follows.

$$\begin{aligned}
B_1 B_2 \sqcup B'_1 B'_2 &= (B_1 \sqcup B'_1) (B_2 \sqcup B'_2) \\
\mathbf{case} B_1 \mathbf{ of} \mathbf{Inl} x \Rightarrow B_2; \mathbf{Inr} y \Rightarrow B_3 \sqcup \mathbf{case} B'_1 \mathbf{ of} \mathbf{Inl} x \Rightarrow B'_2; \mathbf{Inr} y \Rightarrow B'_3 &= \\
\mathbf{case} (B_1 \sqcup B'_1) \mathbf{ of} \mathbf{Inl} x \Rightarrow (B_2 \sqcup B'_2); \mathbf{Inr} y \Rightarrow (B_3 \sqcup B'_3) & \\
\oplus_{\mathbf{p}} l B \sqcup \oplus_{\mathbf{p}} l B' &= \oplus_{\mathbf{p}} l (B \sqcup B') \\
\&\{\ell_i : B_i\}_{i \in I} \sqcup \&\{\ell_j : B'_j\}_{j \in J} &\{\{\ell_k : B_k \sqcup B'_k\}_{k \in I \cup J} \cup \{\ell_i : B_i\}_{i \in I \setminus J} \cup \{\ell_j : B'_j\}_{j \in J \setminus I}\} & \\
x \sqcup x = x \quad \lambda x : T. B \sqcup \lambda x : T. B' &= \lambda x : T. (B \sqcup B')
\end{aligned}$$

$$\begin{array}{c}
 \frac{\text{fv}(V) \cap \text{bv}(M) = \emptyset}{\lambda x : T.M \ V \xrightarrow{\tau, \emptyset}_D M[x := V]} \text{[APPABS]} \quad \frac{M \xrightarrow{\ell, P}_D M'}{\lambda x : T.M \xrightarrow{\lambda, P}_D \lambda x : T.M'} \text{[INABS]} \\
 \\
 \frac{M \xrightarrow{\ell, P}_D M' \quad \ell = \lambda \Rightarrow P \cap \text{pn}(N) = \emptyset}{M \ N \xrightarrow{\tau, P}_D M' \ N} \text{[APP1]} \\
 \\
 \frac{N \xrightarrow{\tau, P}_D N'}{V \ N \xrightarrow{\tau, P}_D V \ N'} \text{[APP2]} \quad \frac{N \xrightarrow{\tau, P}_D N' \quad P \cap \text{pn}(M) = \emptyset}{M \ N \xrightarrow{\tau, P}_D M \ N'} \text{[APP3]} \\
 \\
 \frac{N \xrightarrow{\tau, P}_D N'}{\text{case } N \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M' \xrightarrow{\tau, P}_D \text{case } N' \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M'} \text{[CASE]} \\
 \\
 \frac{M_1 \xrightarrow{\ell, P}_D M'_1 \quad M_2 \xrightarrow{\ell, P}_D M'_2 \quad P \cap \text{pn}(N) = \emptyset}{\text{case } N \text{ of } \text{Inl } x \Rightarrow M_1; \text{Inr } x' \Rightarrow M_2 \xrightarrow{\ell, P}_D \text{case } N' \text{ of } \text{Inl } x \Rightarrow M'_1; \text{Inr } x' \Rightarrow M'_2} \text{[INCASE]} \\
 \\
 \frac{}{\text{case } \text{Inl } V \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M' \xrightarrow{\tau, \emptyset}_D M[x := V]} \text{[CASEL]} \\
 \\
 \frac{}{\text{case } \text{Inr } V \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M' \xrightarrow{\tau, \emptyset}_D M'[x' := V]} \text{[CASER]} \\
 \\
 \frac{}{\text{fst } \text{Pair } V \ V' \xrightarrow{\tau, \emptyset}_D V} \text{[PROJ1]} \quad \frac{}{\text{snd } \text{Pair } V \ V' \xrightarrow{\tau, \emptyset}_D V'} \text{[PROJ2]} \\
 \\
 \frac{D(f(\vec{p}')) = M}{f(\vec{p}) \xrightarrow{\tau, \emptyset}_D M[\vec{p}' := \vec{p}]} \text{[DEF]} \\
 \\
 \frac{\text{fv}(V) = \emptyset}{\text{com}_{q,p} \ V \xrightarrow{\tau, \{q,p\}}_D V[q := p]} \text{[COM]} \quad \frac{}{\text{select}_{q,p} \ l \ M \xrightarrow{\tau, \{q,p\}}_D M} \text{[SEL]} \\
 \\
 \frac{M \xrightarrow{\ell, P}_D M' \quad P \cap \{q, p\} = \emptyset}{\text{select}_{q,p} \ \ell \ M \xrightarrow{\ell, P}_D \text{select}_{q,p} \ \ell \ M'} \text{[INSEL]} \quad \frac{M \rightsquigarrow^* N \quad N \xrightarrow{\tau, P}_D N'}{M \xrightarrow{\tau, P}_D M'} \text{[STR]}
 \end{array}$$

 ■ Figure 8 Semantics of Chor λ .

$$\begin{array}{l}
 \text{fst} \sqcup \text{fst} = \text{fst} \quad \text{snd} \sqcup \text{snd} = \text{snd} \\
 \text{Inl } L \sqcup \text{Inl } L' = \text{Inl } (L \sqcup L') \quad \text{Inr } L \sqcup \text{Inr } L' = \text{Inr } (L \sqcup L') \\
 \text{Pair } L_1 \ L_2 \sqcup \text{Pair } L'_1 \ L'_2 = \text{Pair } (L_1 \sqcup L'_1) \ (L_2 \sqcup L'_2) \quad f \sqcup f = f \\
 \text{recv}_p \sqcup \text{recv}_p = \text{recv}_p \quad \text{send}_p \sqcup \text{send}_p = \text{send}_p \quad \perp \sqcup \perp = \perp
 \end{array}$$

► **Definition 35** (Context). We define a context $C[]$ in Chor λ as follows:

$$\begin{array}{l}
 C[] ::= [] \mid M \ C[] \mid C[] \ M \mid \text{select}_{p,p} \ l \ C[] \mid \text{case } C[] \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x \Rightarrow M \\
 \mid \text{case } M \text{ of } \text{Inl } x \Rightarrow C[]; \text{Inr } x \Rightarrow M \mid \text{case } M \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow C[] \\
 \mid \lambda x : T.C[]
 \end{array}$$

$$\begin{array}{c}
\frac{x \notin \text{fv}(M')}{((\lambda x : T.M) N) M' \rightsquigarrow (\lambda x : T.(M M')) N} \text{[R-ABSR]} \\
\frac{x \notin \text{fv}(M') \quad \text{spn}(M') \cap \text{pn}(N) = \emptyset}{M' ((\lambda x : T.M) N) \rightsquigarrow (\lambda x : T.(M' M)) N} \text{[R-ABSL]} \\
\frac{x, x' \notin \text{fv}(M)}{(\text{case } N \text{ of } \mathbf{Inl} \ x \Rightarrow M_1; \mathbf{Inr} \ x' \Rightarrow M_2) M \rightsquigarrow \text{case } N \text{ of } \mathbf{Inl} \ x \Rightarrow (M_1 M); \mathbf{Inr} \ x' \Rightarrow (M_2 M)} \text{[R-CASER]} \\
\frac{x, x' \notin \text{fv}(M) \quad \text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\text{case } N \text{ of } \mathbf{Inl} \ x \Rightarrow M_1; \mathbf{Inr} \ x' \Rightarrow M_2) \rightsquigarrow \text{case } N \text{ of } \mathbf{Inl} \ x \Rightarrow (M M_1); \mathbf{Inr} \ x' \Rightarrow (M M_2)} \text{[R-CASEL]} \\
\frac{}{(\mathbf{select}_{q,p} \ l \ N) M \rightsquigarrow \mathbf{select}_{q,p} \ l \ (N M)} \text{[R-SELR]} \\
\frac{\text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\mathbf{select}_{q,p} \ l \ N) \rightsquigarrow \mathbf{select}_{q,p} \ l \ (M N)} \text{[R-SELL]} \\
\frac{y \text{ fresh for } M}{\lambda x : T.M \rightsquigarrow \lambda y : T.M[x := y]} \text{[R-ALPH]}
\end{array}$$

■ **Figure 9** Rewriting of Chor λ .

$$\begin{array}{c}
 \frac{\text{fv}(L) = \emptyset}{\text{send}_p L \xrightarrow{\text{send}_p L} \perp} \text{[NSEND]} \quad \frac{}{\text{recv}_p \perp \xrightarrow{\text{recv}_p L} L} \text{[NRECV]} \\
 \frac{B \xrightarrow{\text{send}_q L} \mathbb{D}(q) B'_1 \quad B_2 \xrightarrow{\text{recv}_p L} \mathbb{D}(p) B'_2}{\text{q}[B_1] \mid \text{p}[B_2] \xrightarrow{\tau_{q,p}} \text{q}[B'_1] \mid \text{p}[B'_2]} \text{[NCOM]} \\
 \frac{}{\oplus_p l B \xrightarrow{\oplus_p l} B} \text{[NCHO]} \quad \frac{}{\&_p \{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\&_p \ell_i} B_i} \text{[NOFF]} \\
 \frac{B_i \xrightarrow{\mu} B'_i \text{ for } 1 \leq i \leq n \quad \mu \in \{\tau, \lambda\}}{\&_p \{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\mu} \&_p \{\ell_1 : B'_1, \dots, \ell_n : B'_n\}} \text{[NOFF2]} \\
 \frac{B \xrightarrow{\mu} B' \quad \mu \in \{\tau, \lambda\}}{\oplus_p l B \xrightarrow{\mu} \oplus_p l B'} \text{[NCHO2]} \quad \frac{B_1 \xrightarrow{\oplus_p \ell} \mathbb{D}(q) B'_1 \quad B_2 \xrightarrow{\&_q \ell} \mathbb{D}(p) B'_2}{\text{q}[B_1] \mid \text{p}[B_2] \xrightarrow{\tau_{q,p}} \text{q}[B'_1] \mid \text{p}[B'_2]} \text{[NSEL]} \\
 \frac{}{(\lambda x : T.B) L \xrightarrow{\tau} B[x := L]} \text{[NABSAPP]} \quad \frac{B \xrightarrow{\mu} B' \quad \mu \in \{\tau, \lambda\}}{\lambda x : T.B \xrightarrow{\lambda} \lambda x : T.B'} \text{[NINABS]} \\
 \frac{B \xrightarrow{\mu} B'' \quad \text{if } \mu = \lambda \text{ then } \mu' = \tau \text{ else } \mu' = \mu}{B B' \xrightarrow{\mu'} B'' B'} \text{[NAPP1]} \\
 \frac{B \xrightarrow{\mu} B'}{L B \xrightarrow{\mu} L B'} \text{[NAPP2]} \quad \frac{B' \xrightarrow{\tau} B''}{B B' \xrightarrow{\tau} B B''} \text{[NAPP3]} \\
 \frac{B \xrightarrow{\mu} B''}{\text{case } B \text{ of } \text{Inl } x \Rightarrow B'; \text{Inr } x' \Rightarrow B'' \xrightarrow{\mu} \text{case } B''' \text{ of } \text{Inl } x \Rightarrow B'; \text{Inr } x' \Rightarrow B''} \text{[NCASE]} \\
 \frac{B_1 \xrightarrow{\mu} B'_1 \quad B_2 \xrightarrow{\mu} B'_2 \quad \mu \in \{\lambda, \tau\}}{\text{case } B \text{ of } \text{Inl } x \Rightarrow B_1; \text{Inr } x' \Rightarrow B_2 \xrightarrow{\mu} \text{case } B \text{ of } \text{Inl } x \Rightarrow B'_1; \text{Inr } x' \Rightarrow B'_2} \text{[NCASE2]} \\
 \frac{}{\text{case } \text{Inl } L \text{ of } \text{Inl } x \Rightarrow B; \text{Inr } x' \Rightarrow B' \xrightarrow{\tau} B[x := L]} \text{[NCASEL]} \\
 \frac{}{\text{case } \text{Inr } L \text{ of } \text{Inl } x \Rightarrow B; \text{Inr } x' \Rightarrow B' \xrightarrow{\tau} B'[x' := L]} \text{[NCASER]} \\
 \frac{}{\text{fst Pair } L L' \xrightarrow{\tau} L} \text{[NPROJ1]} \quad \frac{}{\text{snd Pair } L L' \xrightarrow{\tau} L'} \text{[NPROJ2]} \\
 \frac{B \xrightarrow{\tau} \mathbb{D}(p) B'}{\text{p}[B] \xrightarrow{\tau_p} \text{p}[B']} \text{[NPRO]} \quad \frac{\mathcal{N} \xrightarrow{\tau_p} \mathcal{N}''}{\mathcal{N} \mid \mathcal{N}' \xrightarrow{\tau_p} \mathcal{N}'' \mid \mathcal{N}'} \text{[NPAR]} \\
 \frac{D(f(\vec{p}')) = B}{f(\vec{p}) \xrightarrow{\tau} B[\vec{p}' := \vec{p}]} \text{[NFUN]} \quad \frac{B \rightsquigarrow^* B'' \quad B'' \xrightarrow{\mu} B'}{B \xrightarrow{\mu} B'} \text{[NSTR]}
 \end{array}$$

■ Figure 10 Semantics of networks.

$$\begin{array}{c}
\frac{}{((\lambda x.B) B') B'' \rightsquigarrow (\lambda x.B B'') B')} \text{[LR-ABSR]} \\
\frac{\text{pn}(B'') = \emptyset}{B'' ((\lambda x.B) B') \rightsquigarrow (\lambda x.B'' B) B'} \text{[LR-ABSL]} \\
\hline
\frac{(\text{case } B \text{ of } \text{Inl } x \Rightarrow B_1; \text{Inr } x \Rightarrow B_2) B' \rightsquigarrow \text{case } B \text{ of } \text{Inl } x \Rightarrow (B_1 B'); \text{Inr } x \Rightarrow (B_2 B')}{\text{pn}(B') = \emptyset} \text{[LR-CASER]} \\
\hline
\frac{B' (\text{case } B \text{ of } \text{Inl } x \Rightarrow B_1; \text{Inr } x \Rightarrow B_2) \rightsquigarrow \text{case } B \text{ of } \text{Inl } x \Rightarrow (B' B_1); \text{Inr } x \Rightarrow (B' B_2)}{\text{pn}(B) = \emptyset} \text{[LR-CASEL]} \\
\hline
\frac{B (\&_p\{l_1 : B_1, \dots, l_n : B_n\}) \rightsquigarrow \&_p\{l_1 : B B_1, \dots, l_n : B B_n\}}{\text{pn}(B) = \emptyset} \text{[LR-OFFL]} \\
\hline
\frac{(\&_p\{l_1 : B_1, \dots, l_n : B_n\}) B \rightsquigarrow \&_p\{l_1 : B_1 B, \dots, l_n : B_n B\}}{\text{pn}(B') = \emptyset} \text{[LR-OFFR]} \\
\hline
\frac{B' (\oplus_p l B) \rightsquigarrow \oplus_p l (B' B)}{\text{pn}(B') = \emptyset} \text{[LR-CHOL]} \quad \frac{(\oplus_p l B) B' \rightsquigarrow \oplus_p l (B B')}{\text{pn}(B') = \emptyset} \text{[LR-CHOR]} \\
\hline
\frac{}{\perp \perp \rightsquigarrow \perp} \text{[LR-BOTM]} \quad \frac{y \text{ fresh for } B}{\lambda x : T.B \rightsquigarrow \lambda y : T.B[x := y]} \text{[LR-ALPH]}
\end{array}$$

■ **Figure 11** Rewriting of processes.

$$\begin{array}{c}
\frac{\Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash \oplus_p l B : T} \text{[NTCHOR]} \quad \frac{\Sigma; \Gamma \vdash B_i : T \text{ for } 1 \leq i \leq n}{\Sigma; \Gamma \vdash \&_p\{l_1 : B_1, \dots, l_n : B_n\} : T} \text{[NTOFF]} \\
\frac{}{\Sigma; \Gamma \vdash \text{send}_p : T \rightarrow \perp} \text{[NTSEND]} \quad \frac{}{\Sigma; \Gamma \vdash \text{recv}_p : \perp \rightarrow T} \text{[NTRECV]} \\
\frac{\Sigma; \Gamma, x : T \vdash B : T'}{\Sigma; \Gamma \vdash \lambda x : T.B : T \rightarrow T'} \text{[NTABS]} \quad \frac{x : T \in \Gamma}{\Sigma; \Gamma \vdash x : T} \text{[NTVAR]} \\
\frac{\Sigma; \Gamma \vdash B : T \rightarrow T' \quad \Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash B B' : T'} \text{[NTAPP]} \quad \frac{\Sigma; \Gamma \vdash B : \perp \quad \Sigma; \Gamma \vdash B' : \perp}{\Sigma; \Gamma \vdash B B' : \perp} \text{[NTAPP2]} \\
\frac{\Sigma; \Gamma \vdash B : T_1 + T_2 \quad \Sigma; \Gamma, x : T_1 \vdash B' : T \quad \Sigma; \Gamma, x' : T_2 \vdash B'' : T}{\Sigma; \Gamma \vdash \text{case } B \text{ of } \text{Inl } x \Rightarrow B'; \text{Inr } x' \Rightarrow B'' : T} \text{[NTCASE]} \\
\frac{f : T \in \Gamma}{\Sigma; \Gamma \vdash f : T} \text{[NTDEF]} \quad \frac{}{\Sigma; \Gamma \vdash () : ()} \text{[NTUNIT]} \quad \frac{}{\Sigma; \Gamma \vdash \perp : \perp} \text{[NTBOTM]} \\
\frac{}{\Sigma; \Gamma \vdash \text{Pair} : T \rightarrow T' \rightarrow (T \times T')} \text{[NTPAIR]} \\
\frac{}{\Sigma; \Gamma \vdash \text{fst} : (T \times T') \rightarrow T} \text{[NTPROJ1]} \quad \frac{}{\Sigma; \Gamma \vdash \text{snd} : (T \times T') \rightarrow T'} \text{[NTPROJ2]} \\
\frac{\Sigma; \Gamma \vdash B : T' \quad \{T = T', T' = T\} \cap \Sigma \neq \emptyset}{\Sigma; \Gamma \vdash B : T} \text{[NTEQ]} \\
\frac{\forall f \in \text{dom}(\mathbb{D}) \quad f : T \in \Gamma \quad \Sigma; \Gamma \vdash \mathbb{D}(f) : T}{\Sigma; \Gamma \vdash \mathbb{D}} \text{[NTDEFS]}
\end{array}$$

■ **Figure 12** Typing rules for behaviours.

Choreographies:

$$\begin{aligned}
 \llbracket M \ N \rrbracket_{\mathfrak{p}} &= \begin{cases} \llbracket M \rrbracket_{\mathfrak{p}} \ \llbracket N \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(M)) \text{ or } \mathfrak{p} \in \text{pn}(M) \cap \text{pn}(N) \\ \perp & \text{if } \llbracket M \rrbracket_{\mathfrak{p}} = \llbracket N \rrbracket_{\mathfrak{p}} = \perp \\ \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \llbracket N \rrbracket_{\mathfrak{p}} = \perp \\ \llbracket N \rrbracket_{\mathfrak{p}} & \text{otherwise} \end{cases} \\
 \llbracket \lambda x : T.M \rrbracket_{\mathfrak{p}} &= \begin{cases} \lambda x. \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\lambda x : T.M)) \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket \text{case } M \text{ of } \text{Inl } x \Rightarrow N; \text{Inr } x' \Rightarrow N' \rrbracket_{\mathfrak{p}} &= \\
 &\begin{cases} \text{case } \llbracket M \rrbracket_{\mathfrak{p}} \text{ of } \text{Inl } x \Rightarrow \llbracket N \rrbracket_{\mathfrak{p}}; \text{Inr } x' \Rightarrow \llbracket N' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(M)) \\ \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \llbracket N \rrbracket_{\mathfrak{p}} = \llbracket N' \rrbracket_{\mathfrak{p}} = \perp \\ \llbracket N \rrbracket_{\mathfrak{p}} \sqcup \llbracket N' \rrbracket_{\mathfrak{p}} & \text{if } \llbracket M \rrbracket_{\mathfrak{p}} = \perp \\ (\lambda x'' : \perp. \llbracket N \rrbracket_{\mathfrak{p}} \sqcup \llbracket N' \rrbracket_{\mathfrak{p}}) \ \llbracket M \rrbracket_{\mathfrak{p}} & \text{otherwise, for some} \\ & x'' \notin \text{fv}(N) \cup \text{fv}(N') \end{cases} \\
 \llbracket \text{select}_{\mathfrak{q}, \mathfrak{q}'} \ell \ M \rrbracket_{\mathfrak{p}} &= \begin{cases} \oplus_{\mathfrak{q}'} \ell \ \llbracket M \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} = \mathfrak{q} \neq \mathfrak{q}' \\ \&_{\mathfrak{q}} \{ \ell : \llbracket M \rrbracket_{\mathfrak{p}} \} & \text{if } \mathfrak{p} = \mathfrak{q}' \neq \mathfrak{q} \\ \llbracket M \rrbracket_{\mathfrak{p}} & \text{otherwise} \end{cases} \\
 \llbracket \text{com}_{\mathfrak{q}, \mathfrak{q}'} \rrbracket_{\mathfrak{p}} &= \begin{cases} \lambda x.x & \text{if } \mathfrak{p} = \mathfrak{q} = \mathfrak{q}' \\ \text{send}_{\mathfrak{q}'} & \text{if } \mathfrak{p} = \mathfrak{q} \neq \mathfrak{q}' \\ \text{recv}_{\mathfrak{q}} & \text{if } \mathfrak{p} = \mathfrak{q}' \neq \mathfrak{q} \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket () @ \mathfrak{q} \rrbracket_{\mathfrak{p}} &= \begin{cases} () & \text{if } \mathfrak{q} = \mathfrak{p} \\ \perp & \text{otherwise} \end{cases} \quad \llbracket x \rrbracket_{\mathfrak{p}} = \begin{cases} x & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(x)) \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket f(\vec{\mathfrak{p}}) \rrbracket_{\mathfrak{p}} &= \begin{cases} f_i(\mathfrak{p}_1, \dots, \mathfrak{p}_{i-1}, \mathfrak{p}_{i+1}, \dots, \mathfrak{p}_n) & \text{if } \vec{\mathfrak{p}} = \mathfrak{p}_1, \dots, \mathfrak{p}_{i-1}, \mathfrak{p}, \mathfrak{p}_{i+1}, \dots, \mathfrak{p}_n \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket \text{Pair } V \ V' \rrbracket_{\mathfrak{p}} &= \begin{cases} \text{Pair } \llbracket V \rrbracket_{\mathfrak{p}} \ \llbracket V' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(V) \times \text{type}(V')) \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket \text{fst} \rrbracket_{\mathfrak{p}} &= \begin{cases} \text{fst} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\text{fst})) \\ \perp & \text{otherwise} \end{cases} \quad \llbracket \text{snd} \rrbracket_{\mathfrak{p}} = \begin{cases} \text{snd} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\text{snd})) \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket \text{Inl } V \rrbracket_{\mathfrak{p}} &= \begin{cases} \text{Inl } \llbracket V \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\text{Inl } V)) \\ \perp & \text{otherwise} \end{cases} \quad \llbracket \text{Inr } V \rrbracket_{\mathfrak{p}} = \begin{cases} \text{Inr } \llbracket V \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(\text{type}(\text{Inr } V)) \\ \perp & \text{otherwise} \end{cases}
 \end{aligned}$$

Types:

$$\begin{aligned}
 \llbracket T \rightarrow_{\rho} T' \rrbracket_{\mathfrak{p}} &= \begin{cases} \llbracket T \rrbracket_{\mathfrak{p}} \rightarrow \llbracket T' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \rho \cup \text{pn}(T) \cup \text{pn}(T') \\ \perp & \text{otherwise} \end{cases} \quad \llbracket () @ \mathfrak{q} \rrbracket_{\mathfrak{p}} = \begin{cases} () & \text{if } \mathfrak{q} = \mathfrak{p} \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket T \times T' \rrbracket_{\mathfrak{p}} &= \begin{cases} \llbracket T \rrbracket_{\mathfrak{p}} \times \llbracket T' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(T \times T') \\ \perp & \text{otherwise} \end{cases} \quad \llbracket T + T' \rrbracket_{\mathfrak{p}} = \begin{cases} \llbracket T \rrbracket_{\mathfrak{p}} + \llbracket T' \rrbracket_{\mathfrak{p}} & \text{if } \mathfrak{p} \in \text{pn}(T + T') \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket t @ \vec{\mathfrak{p}} \rrbracket_{\mathfrak{p}} &= \begin{cases} t_i & \text{if } \vec{\mathfrak{p}} = \mathfrak{p}_1, \dots, \mathfrak{p}_{i-1}, \mathfrak{p}, \mathfrak{p}_{i+1}, \dots, \mathfrak{p}_n \\ \perp & \text{otherwise} \end{cases}
 \end{aligned}$$

Definitions:

$$\llbracket D \rrbracket = \{ f_i(\mathfrak{p}_1, \dots, \mathfrak{p}_{i-1}, \mathfrak{p}_{i+1}, \dots, \mathfrak{p}_n) \mapsto \llbracket D(f(\mathfrak{p}_1, \dots, \mathfrak{p}_n)) \rrbracket_{\mathfrak{p}_i} \mid f(\mathfrak{p}_1, \dots, \mathfrak{p}_n) \in \text{dom}(D) \}$$

■ **Figure 13** Projecting Chor λ onto a process.

A.1 Proof of Theorem 25

Proof of Lemma 5. Straightforward from the network semantics. \blacktriangleleft

► **Lemma 36.** *Given a value V , if $\Theta; \Sigma; \Gamma \vdash V : T$ and $\llbracket V \rrbracket$ is defined then for any process p in $\text{pn}(V)$, $\llbracket V \rrbracket_p = L$.*

Proof. Straightforward from the projection rules. \blacktriangleleft

► **Lemma 37.** *Given a type T , for any process $p \notin \text{pn}(T)$, $\llbracket T \rrbracket_p = \perp$.*

Proof. Straightforward from induction on T . \blacktriangleleft

► **Lemma 38.** *Given a value V , for any process $p \notin \text{pn}(\text{type}(V))$, if $\llbracket V \rrbracket_p$ is defined then $\llbracket V \rrbracket_p = \perp$.*

Proof. Follows from Lemmas 36 and 37 and the projection rules. \blacktriangleleft

► **Lemma 39.** *If $M \rightsquigarrow M'$ and $M \xrightarrow{\tau, P}_D M''$ and $\llbracket M \rrbracket$ is defined then $M' \xrightarrow{\tau, P}_D M'''$ such that $M'' \rightsquigarrow^* M'''$*

Proof. Follows from case analysis on $M \rightsquigarrow M'$. \blacktriangleleft

► **Lemma 40.** *If $M \rightsquigarrow M'$ then for any process p , $\llbracket M \rrbracket_p \rightsquigarrow \cup \xrightarrow{\tau}^* B$ such that $B \equiv \llbracket M' \rrbracket_p$*

Proof. Follows from case analysis on $M \rightsquigarrow M'$. \blacktriangleleft

Proof of Theorem 25. We prove this by structural induction on $M \xrightarrow{\tau, P}_D M'$.

- Assume $M = \lambda x : T.N V$ and $M' = N[x := V]$. Then for any process $p \in \text{pn}(\text{type}(\lambda x : T.N))$, we have $\llbracket M \rrbracket_p = (\lambda x : \llbracket T \rrbracket_p . \llbracket N \rrbracket_p) \llbracket V \rrbracket_p$ and $\llbracket M' \rrbracket_p = \llbracket N \rrbracket_p[x := \llbracket V \rrbracket_p]$, and for any $p' \notin \text{pn}(\text{type}(\lambda x : T.N))$, we have $p' \notin \text{pn}(\text{type}(V))$ and therefore $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'} = \perp$. We therefore get $p[\llbracket M \rrbracket_p] \xrightarrow{\tau}_{[D]} \llbracket M' \rrbracket_p$ for all $p \in \text{pn}(\text{type}(\lambda x : T.N))$ and define $\mathcal{N} = \prod_{p \in \text{pn}(\text{type}(\lambda x : T.N))} p[\llbracket M' \rrbracket_p] \mid \prod_{p' \notin \text{pn}(\text{type}(\lambda x : T.N))} p'[\perp]$ and the result follows.
- Assume $M = N M''$, $M' = N' M''$, and $N \xrightarrow{\tau, P}_D N'$. Then for any process $p \in \text{pn}(\text{type}(N))$, $\llbracket M \rrbracket_p = \llbracket N \rrbracket_p \llbracket M'' \rrbracket_p$ and $\llbracket M' \rrbracket_p = \llbracket N' \rrbracket_p \llbracket M'' \rrbracket_p$. For any process p' such that $\llbracket N \rrbracket_{p'} = \llbracket M'' \rrbracket_{p'} = \perp$, by induction we have $\llbracket N' \rrbracket_{p'} = \perp$, and therefore $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'} = \perp$. For any other process p'' such that $\llbracket N \rrbracket_{p''} = \perp$, by induction we get $\llbracket N' \rrbracket_{p''} = \perp$ and therefore $\llbracket M \rrbracket_{p''} = \llbracket M' \rrbracket_{p''} = \llbracket M'' \rrbracket_{p''}$. For any other process p''' such that $\llbracket M'' \rrbracket_{p'''} = \perp$, we get $\llbracket M \rrbracket_{p'''} = \llbracket N \rrbracket_{p'''} = \perp$ and $\llbracket M' \rrbracket_{p'''} = \llbracket N' \rrbracket_{p'''} = \perp$. And by induction $\llbracket N \rrbracket \xrightarrow{*_D} \mathcal{N}_N$ and $N' \xrightarrow{*_D} N''$ for $\mathcal{N}_N \equiv \llbracket N'' \rrbracket$. For any process p we therefore get $\llbracket N \rrbracket_p \xrightarrow{\mu_0}_{[D]} \xrightarrow{\mu_1}_{[D]} \dots B_p$ for $B_p \equiv \llbracket N'' \rrbracket_p$ for some sequences of transitions $\xrightarrow{\mu_0}_{[D]} \xrightarrow{\mu_1}_{[D]} \dots$, and from the network semantics we get

$$\llbracket M \rrbracket \rightarrow^* \left(\prod_{p \in \text{pn}(\text{type}(N)) \cup (\text{pn}(N) \cap \text{pn}(M''))} p[B_p \llbracket M'' \rrbracket_p] \mid \prod_{\llbracket N \rrbracket_{p'} = \llbracket M'' \rrbracket_{p'} = \perp} p'[\perp] \right) \mid \left(\prod_{\llbracket M \rrbracket_{p''} = \llbracket M'' \rrbracket_{p''}} p''[\llbracket M'' \rrbracket_{p''}] \mid \prod_{\llbracket M \rrbracket_{p'''} = \llbracket N \rrbracket_{p'''}} p''[B_{p''}] \right) = \mathcal{N}$$

and $M' \rightarrow^* N'' M$. And since $\llbracket N \rrbracket \xrightarrow{*_D} \mathcal{N}'$ and $\llbracket N' \rrbracket \xrightarrow{*_D} \mathcal{N}'_N$, we know these sequences of transitions can synchronise when necessary, and if $\llbracket N \rrbracket_{p'''} \neq \llbracket N' \rrbracket_{p'''} = \perp$ then we can do the extra application to get rid of this unit.

- Assume $M = V N$, $M' = V N'$, and $N \xrightarrow{\tau, P}_D N'$. Then for any process $\mathfrak{p} \in \text{pn}(\text{type}(V))$, $\llbracket M \rrbracket_{\mathfrak{p}} = \llbracket V \rrbracket_{\mathfrak{p}} \llbracket N \rrbracket_{\mathfrak{p}}$ and $\llbracket M' \rrbracket_{\mathfrak{p}} = \llbracket V \rrbracket_{\mathfrak{p}} \llbracket N' \rrbracket_{\mathfrak{p}}$. Since V is a value, for any process $\mathfrak{p}' \notin \text{pn}(\text{type}(V))$, we have $\llbracket V \rrbracket_{\mathfrak{p}'} = \perp$ and so for any process \mathfrak{p}' such that $\llbracket V \rrbracket_{\mathfrak{p}'} = \llbracket N \rrbracket_{\mathfrak{p}'} = \perp$, by induction we get $\llbracket N' \rrbracket_{\mathfrak{p}'} = \perp$ and therefore $\llbracket M \rrbracket_{\mathfrak{p}'} = \llbracket M' \rrbracket_{\mathfrak{p}'} = \perp$. For any other process \mathfrak{p}'' such that $\llbracket V \rrbracket_{\mathfrak{p}''} = \perp$, we have $\llbracket M \rrbracket_{\mathfrak{p}''} = \llbracket N \rrbracket_{\mathfrak{p}''}$ and $\llbracket M' \rrbracket_{\mathfrak{p}''} = \llbracket N' \rrbracket_{\mathfrak{p}''}$. By induction, $\llbracket N \rrbracket \xrightarrow{*_D} \mathcal{N}_N$ and $N' \xrightarrow{*_D} \mathcal{N}'$ for $\mathcal{N}' \sqsupseteq \llbracket N'' \rrbracket$. For any process \mathfrak{p} we therefore get $\llbracket N \rrbracket_{\mathfrak{p}} \xrightarrow{\mu_0}_{[D](\mathfrak{p})} \xrightarrow{\mu_1}_{[D](\mathfrak{p})} \dots B_{\mathfrak{p}}$ for $B_{\mathfrak{p}} \sqsupseteq \llbracket N'' \rrbracket_{\mathfrak{p}}$ for some sequences of transitions $\xrightarrow{\mu_0}_{[D](\mathfrak{p})} \xrightarrow{\mu_1}_{[D](\mathfrak{p})} \dots$ and from the network semantics we get

$$\llbracket M \rrbracket \xrightarrow{*} \prod_{\mathfrak{p} \in \text{pn}(\text{type}(N))} \mathfrak{p}[\llbracket V \rrbracket_{\mathfrak{p}} B_{\mathfrak{p}}] \mid \prod_{\mathfrak{p}' \notin \text{pn}(\text{type}(N))} \mathfrak{p}'[B_{\mathfrak{p}'}] = \mathcal{N}$$

and

$$M' \xrightarrow{*} V N''$$

and the result follows.

- Assume $M = M'' N$, $M' = M'' N'$, $N \xrightarrow{\tau, P}_D N'$, and $\text{pn}(M) \cap P = \emptyset$. Then for any $\mathfrak{p} \in P$, $\text{pn}(\llbracket M'' \rrbracket_{\mathfrak{p}}) \cap P = \emptyset$ and the result follows from induction and using rule NAPP3.
- Assume $M = \mathbf{case} N \mathbf{ of} \mathbf{Inl} x \Rightarrow N'; \mathbf{Inr} x' \Rightarrow N''$, $M' = \mathbf{case} M'' \mathbf{ of} \mathbf{Inl} x \Rightarrow N'; \mathbf{Inr} x' \Rightarrow N''$, and $N \xrightarrow{\tau, P}_D M''$. Then for any process \mathfrak{p} such that $\mathfrak{p} \in \text{pn}(\text{type}(N))$, we have projections $\llbracket M \rrbracket_{\mathfrak{p}} = \mathbf{case} \llbracket N \rrbracket_{\mathfrak{p}} \mathbf{ of} \mathbf{Inl} x \Rightarrow \llbracket N' \rrbracket_{\mathfrak{p}}; \mathbf{Inr} x' \Rightarrow \llbracket N'' \rrbracket_{\mathfrak{p}}$ and $\llbracket M' \rrbracket_{\mathfrak{p}} = \mathbf{case} \llbracket M'' \rrbracket_{\mathfrak{p}} \mathbf{ of} \mathbf{Inl} x \Rightarrow \llbracket N' \rrbracket_{\mathfrak{p}}; \mathbf{Inr} x' \Rightarrow \llbracket N'' \rrbracket_{\mathfrak{p}}$. For any other process \mathfrak{p}' such that $\llbracket N \rrbracket_{\mathfrak{p}'} = \llbracket N' \rrbracket_{\mathfrak{p}'} = \llbracket N'' \rrbracket_{\mathfrak{p}'} = \perp$, by induction we get $\llbracket M'' \rrbracket_{\mathfrak{p}'} = \perp$, and therefore $\llbracket M \rrbracket_{\mathfrak{p}'} = \llbracket M' \rrbracket_{\mathfrak{p}'} = \perp$. For any other process \mathfrak{p}'' such that $\llbracket N \rrbracket_{\mathfrak{p}''} = \perp$, we get $\llbracket M \rrbracket_{\mathfrak{p}''} = \llbracket M' \rrbracket_{\mathfrak{p}''} = \llbracket N' \rrbracket_{\mathfrak{p}''} \sqcup \llbracket N'' \rrbracket_{\mathfrak{p}''}$. For any other processes \mathfrak{p}''' such that $\llbracket N' \rrbracket_{\mathfrak{p}'''} = \llbracket N'' \rrbracket_{\mathfrak{p}'''} = \perp$, we have $\llbracket M \rrbracket_{\mathfrak{p}'''} = \llbracket N \rrbracket_{\mathfrak{p}'''} = \llbracket M' \rrbracket_{\mathfrak{p}'''} = \llbracket M'' \rrbracket_{\mathfrak{p}'''}$. For any other process \mathfrak{p}'''' , we have $\llbracket M \rrbracket_{\mathfrak{p}''''} = (\lambda x : \perp. \llbracket N' \rrbracket_{\mathfrak{p}''''} \sqcup \llbracket N'' \rrbracket_{\mathfrak{p}''''}) \llbracket N \rrbracket_{\mathfrak{p}''''}$ and $\llbracket M' \rrbracket_{\mathfrak{p}''''} = (\lambda x. \llbracket N' \rrbracket_{\mathfrak{p}''''} \sqcup \llbracket N'' \rrbracket_{\mathfrak{p}''''}) \llbracket M'' \rrbracket_{\mathfrak{p}''''}$ for $x \notin \text{fv}(N') \cup \text{fv}(N'')$. The rest follows by simple induction similar to the second case.
- Assume $M = \mathbf{case} N \mathbf{ of} \mathbf{Inl} x \Rightarrow N_1; \mathbf{Inr} x' \Rightarrow N_2$, $M' = \mathbf{case} N \mathbf{ of} \mathbf{Inl} x \Rightarrow N'_1; \mathbf{Inr} x' \Rightarrow N'_2$, $N_1 \xrightarrow{\tau, P}_D N'_1$, $N_2 \xrightarrow{\tau, P}_D N'_2$, and $P \cap \text{pn}(N) = \emptyset$. Then for any process \mathfrak{p} such that $\mathfrak{p} \in \text{pn}(\text{type}(N))$, we have $\llbracket M \rrbracket_{\mathfrak{p}} = \mathbf{case} \llbracket N \rrbracket_{\mathfrak{p}} \mathbf{ of} \mathbf{Inl} x \Rightarrow \llbracket N_1 \rrbracket_{\mathfrak{p}}; \mathbf{Inr} x' \Rightarrow \llbracket N_2 \rrbracket_{\mathfrak{p}}$. For any other process \mathfrak{p}' such that $\llbracket N \rrbracket_{\mathfrak{p}'} = \llbracket N_1 \rrbracket_{\mathfrak{p}'} = \llbracket N_2 \rrbracket_{\mathfrak{p}'} = \perp$, by induction we get $\llbracket N'_1 \rrbracket_{\mathfrak{p}'} = \llbracket N'_2 \rrbracket_{\mathfrak{p}'} = \perp$, and therefore $\llbracket M \rrbracket_{\mathfrak{p}'} = \llbracket M' \rrbracket_{\mathfrak{p}'} = \perp$. For any other process \mathfrak{p}'' such that $\llbracket N \rrbracket_{\mathfrak{p}''} = \perp$, we get $\llbracket M \rrbracket_{\mathfrak{p}''} = \llbracket N_1 \rrbracket_{\mathfrak{p}''} \sqcup \llbracket N_2 \rrbracket_{\mathfrak{p}''}$. For any other processes \mathfrak{p}''' such that $\llbracket N_1 \rrbracket_{\mathfrak{p}'''} = \llbracket N_2 \rrbracket_{\mathfrak{p}'''} = \perp$, we have $\llbracket M \rrbracket_{\mathfrak{p}'''} = \llbracket N \rrbracket_{\mathfrak{p}'''}$. For any other process \mathfrak{p}'''' , we have $\llbracket M \rrbracket_{\mathfrak{p}''''} = (\lambda x : \perp. \llbracket N_1 \rrbracket_{\mathfrak{p}''''} \sqcup \llbracket N_2 \rrbracket_{\mathfrak{p}''''}) \llbracket N \rrbracket_{\mathfrak{p}''''}$. If $\llbracket N'_1 \rrbracket_{\mathfrak{p}} \sqcup \llbracket N'_2 \rrbracket_{\mathfrak{p}}$ is defined for all \mathfrak{p} then the result follows from induction. Otherwise we have M_1 and M_2 such that $N'_1 \xrightarrow{\tau, P}_D M_1$ and $N'_2 \xrightarrow{\tau, P}_D M_2$ and $\llbracket M_1 \rrbracket_{\mathfrak{p}} \sqcup \llbracket M_2 \rrbracket_{\mathfrak{p}}$ for all \mathfrak{p} , and the result follows from induction on these transitions.
- Assume $M = \mathbf{case} \mathbf{Inl} V \mathbf{ of} \mathbf{Inl} x \Rightarrow N; \mathbf{Inr} x' \Rightarrow N'$ and $M' = N[x := V]$. Then for any process $\mathfrak{p} \in \text{pn}(\text{type}(\mathbf{Inl} V))$, we have $\llbracket M \rrbracket_{\mathfrak{p}} = \mathbf{case} \mathbf{Inl} \llbracket V \rrbracket_{\mathfrak{p}} \mathbf{ of} \mathbf{Inl} x \Rightarrow \llbracket N \rrbracket_{\mathfrak{p}}; \mathbf{Inr} x' \Rightarrow \llbracket N' \rrbracket_{\mathfrak{p}}$ and $\llbracket M' \rrbracket_{\mathfrak{p}} = \llbracket N[x := \llbracket V \rrbracket_{\mathfrak{p}}] \rrbracket_{\mathfrak{p}}$. By Lemma 38, $\llbracket N[x := \llbracket V \rrbracket_{\mathfrak{p}}] \rrbracket_{\mathfrak{p}} = \llbracket N \rrbracket_{\mathfrak{p}}[x := \llbracket V \rrbracket_{\mathfrak{p}}]$. For any other process $\mathfrak{p}' \notin \text{pn}(\text{type}(\mathbf{Inl} V))$, $\llbracket \mathbf{Inl} V \rrbracket_{\mathfrak{p}'} = \perp$, and therefore $\llbracket M \rrbracket_{\mathfrak{p}'} = \llbracket N \rrbracket_{\mathfrak{p}'} \sqcup \llbracket N' \rrbracket_{\mathfrak{p}'} \sqsupset \llbracket N \rrbracket_{\mathfrak{p}'} = \llbracket M' \rrbracket_{\mathfrak{p}'}$. The result follows.
- Assume $M = \mathbf{case} \mathbf{Inr} V \mathbf{ of} \mathbf{Inl} x \Rightarrow N; \mathbf{Inr} x' \Rightarrow N'$ and $M' = N'[x' := V]$. This case is similar to the previous.
- Assume $M = \mathbf{case} N \mathbf{ of} \mathbf{Inl} x \Rightarrow N_1; \mathbf{Inr} x' \Rightarrow N_2$, $M' = \mathbf{case} N \mathbf{ of} \mathbf{Inl} x \Rightarrow N'_1; \mathbf{Inr} x' \Rightarrow N'_2$, $N_1 \xrightarrow{P}_D N'_1$, $N_2 \xrightarrow{P}_D N'_2$, and $P \cap \text{pn}(N) = \emptyset$. This case is similar to case four.

- Assume $M = \mathbf{com}_{q,p}V$ and $M' = V[q := p]$ and $\text{fv}(V) = \emptyset$. Then if $q \neq p$, $\llbracket M \rrbracket_p = \mathbf{recv}_q \perp$, $\llbracket M' \rrbracket_p = \llbracket V[q := p] \rrbracket_p = \llbracket V \rrbracket_p[q := p]$ since $\text{pn}(\text{type}(V)) = q$, $\llbracket M \rrbracket_q = \mathbf{send}_p \llbracket V \rrbracket_q$, $\llbracket M' \rrbracket_q = \perp$, and for any $p' \notin \{q, p\}$, $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'} = \perp$. We therefore get $\llbracket M \rrbracket_p \xrightarrow{\mathbf{recv}_q \llbracket V \rrbracket_q[q := p]}_{[D]} \llbracket M' \rrbracket_p$, $\llbracket M \rrbracket_q \xrightarrow{\mathbf{send}_p \llbracket V \rrbracket_q}_{[D]} \llbracket M' \rrbracket_q$, and $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'}$. We define $\mathcal{N} = \mathcal{N}' = \llbracket M' \rrbracket$ and the result follows. If $q = p$, then $\llbracket M \rrbracket_p = (\lambda x.x) \llbracket V \rrbracket_p$ and $\llbracket M' \rrbracket_p = \llbracket V \rrbracket_p$ and $\mathcal{N} = \mathcal{N}' = \llbracket M' \rrbracket$ and the result follows.
- Assume $M = \mathbf{select}_{q,p} l M'$. Then $\llbracket M \rrbracket_q = \oplus_p l \llbracket M' \rrbracket_q$, $\llbracket M \rrbracket_p = \&\{l : \llbracket M' \rrbracket_p\}$, and for any $p' \notin \{q, p\}$, $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'}$. We therefore get $\llbracket M \rrbracket \xrightarrow{\tau_{p,q}}_{[D]} \llbracket M \rrbracket \setminus \{p, q\} \mid p[\llbracket M' \rrbracket_p] \mid q[\llbracket M' \rrbracket_q]$ and the result follows.
- Assume $M = \mathbf{select}_{q,p} l N$, $M' = \mathbf{select}_{q,p} l N'$, $N \xrightarrow{\tau, P}_D N'$, and $P \cap \{q, p\} = \emptyset$. Then $\llbracket M \rrbracket_q = \oplus_p l \llbracket N \rrbracket_q$, $\llbracket M' \rrbracket_q = \oplus_p l \llbracket N' \rrbracket_q$, $\llbracket M \rrbracket_p = \&\{l : \llbracket N \rrbracket_p\}$, $\llbracket M' \rrbracket_p = \&\{l : \llbracket N' \rrbracket_p\}$, and for any $p' \notin \{q, p\}$, $\llbracket M \rrbracket_{p'} = \llbracket N \rrbracket_{p'}$ and $\llbracket M' \rrbracket_{p'} = \llbracket N' \rrbracket_{p'}$. The result follows from induction and using rules NOFF2 and NCHO2.
- Assume $M = \mathbf{fst Pair} V V'$ and $M' = V$. Then for any process $p \in \text{pn}(\text{type}(\mathbf{Pair} M' V'))$, $\llbracket M \rrbracket_p = \mathbf{fst Pair} \llbracket M' \rrbracket_p \llbracket V' \rrbracket_p$ and for any other process $p' \notin \text{pn}(\text{type}(\mathbf{Pair} M' V'))$, we have $\llbracket M \rrbracket_{p'} = \perp$ and $\llbracket M' \rrbracket_{p'} = \perp$. We define $\mathcal{N} = \mathcal{N}' = \llbracket M' \rrbracket$ and the result follows.
- Assume $M = \mathbf{snd Pair} V V'$ and $M' = V'$. Then the case is similar to the previous.
- Assume $M = f(\vec{p})$ and $M' = D(f(\vec{p}'))[\vec{p}' := \vec{p}]$. Then the result follows from the definition of $[D]$.
- Assume there exists N such that $M \rightsquigarrow N$ and $N \xrightarrow{\tau, P}_D M'$. Then the result follows from induction and Lemma 40. ◀

A.2 Proof of Theorem 26

► **Definition 41.** Given a network $\mathcal{N} = \prod_{p \in \rho} p[B_p]$, we have $\mathcal{N} \setminus \rho' = \prod_{p \in (\rho \setminus \rho')} p[B_p]$

► **Lemma 42.** For any process p and network \mathcal{N} , if $\mathcal{N} \xrightarrow{\tau, P} \mathcal{N}'$ and $p \notin P$ then $\mathcal{N}(p) = \mathcal{N}'(p)$.

Proof. Straightforward from the network semantics. ◀

► **Lemma 43.** For any set of processes P and network \mathcal{N} , if $\mathcal{N} \xrightarrow{\tau, P'} \mathcal{N}'$ and $P \cap P' = \emptyset$ then $\mathcal{N} \setminus P \xrightarrow{\tau, P'} \mathcal{N}' \setminus P$.

Proof. Straightforward from the network semantics. ◀

► **Lemma 44.** If $\llbracket M \rrbracket_p \rightsquigarrow B$ then there exists M' such that $M \rightsquigarrow M'$ and $B \equiv \llbracket M' \rrbracket_p$

Proof. Follows from case analysis on $\llbracket M \rrbracket_p \rightsquigarrow B$ keeping in mind that $\llbracket M \rrbracket_p$ cannot be $\perp \perp$. ◀

Proof of Theorem 26. If $\llbracket M \rrbracket \xrightarrow{*_D} \mathcal{N}$ uses rule NSTR then this follows from Lemma 44. Otherwise we prove this by structural induction on M .

- Assume $M = N_1 N_2$. Then for any process $p \in \text{pn}(\text{type}(N_1)) \cup (\text{pn}(N_1) \cap \text{pn}(N_2))$, $\llbracket M \rrbracket_p = \llbracket N_1 \rrbracket_p \llbracket N_2 \rrbracket_p$, for any process p' such that $\llbracket N_1 \rrbracket_{p'} = \llbracket N_2 \rrbracket_{p'} = \perp$, we have $\llbracket M \rrbracket_{p'} = \perp$. For any other process p'' such that $\llbracket N_1 \rrbracket_{p''} = \perp$, $\llbracket M \rrbracket_{p''} = \llbracket N_2 \rrbracket_{p''}$. For any other process p''' such that $\llbracket N_2 \rrbracket_{p'''} = \perp$, we get $\llbracket M \rrbracket_{p'''} = \llbracket N_1 \rrbracket_{p'''}$. We then have 2 cases.
 - Assume $N_2 = V$. Then $\llbracket N_2 \rrbracket_p = L$ by Lemma 36, and for any p' such that $p' \notin \text{pn}(\text{type}(N_2)) \subseteq \text{pn}(\text{type}(N_1))$, by Lemma 38, $\llbracket N_2 \rrbracket_{p'} = \perp$ and therefore $\llbracket M \rrbracket_{p'} = \llbracket N_1 \rrbracket_{p'}$, and we have 5 cases.

* Assume $N_1 = \lambda x : T.N_3$. Then for any process $\mathfrak{p} \in \text{pn}(\text{type}(N_1))$, $\llbracket N_1 \rrbracket_{\mathfrak{p}} = \lambda x : \llbracket T \rrbracket_{\mathfrak{p}} . \llbracket N_3 \rrbracket_{\mathfrak{p}}$. And for any process $\mathfrak{p}' \notin \text{pn}(\text{type}(N_1))$, $\llbracket N_1 \rrbracket_{\mathfrak{p}'} = \perp$. We have two cases, using either rule NABSAPP or rules NINABS and NAPP1.

If we use rule NABSAPP, then there exists \mathfrak{p}'' such that $\mathfrak{P} = \mathfrak{p}''$ and $\mathfrak{p}'' \in \text{pn}(\text{type}(N_1))$. We then get $\llbracket M \rrbracket \xrightarrow{\tau, \mathfrak{P}}_{[D]} \mathcal{M} = \llbracket M \rrbracket \setminus \{\mathfrak{p}''\} \mid \mathfrak{p}''[\llbracket N_3 \rrbracket_{\mathfrak{p}''} [x := \llbracket N_2 \rrbracket_{\mathfrak{p}''}]]$. Since $\mathcal{M} \rightarrow^* \llbracket N_3 [x := N_2] \rrbracket$ and the remaining transitions in $\llbracket M \rrbracket \rightarrow^*_{[D]} \mathcal{N}$ take place in N_3 , the result follows from using rule NABSAPP in every process in $\text{pn}(\text{type}(N_1))$ and induction.

If we use rules NINABS and NAPP1 then there exists \mathfrak{p}'' such that $\mathfrak{P} = \mathfrak{p}''$ and $\llbracket N_3 \rrbracket_{\mathfrak{p}''} \xrightarrow{\mu} B$ and

$$\llbracket M \rrbracket \xrightarrow{\mu}_{[D]} \llbracket M \rrbracket \setminus \{\mathfrak{p}''\} \mid \mathfrak{p}''[\lambda x. B \llbracket N_2 \rrbracket_{\mathfrak{p}''}] \rightarrow^*_{[D]} \mathcal{N}$$

By induction, $N_3 \rightarrow^*_D N'_3$ and $(\llbracket N_3 \rrbracket \setminus \{\mathfrak{p}''\} \mid \mathfrak{p}''[B] \rightarrow_{\mathbb{D}} \mathcal{N}'')$ such that $\llbracket N_3 \rrbracket \supseteq \mathcal{N}''$, and we define $M' = \lambda x : T.N'_3 N_2$ and

$$\mathcal{N}' = (\mathcal{N} \setminus \text{pn}(\text{type}(N_1))) \mid \prod_{\mathfrak{p} \in \text{pn}(\text{type}(N_3))} \mathfrak{p}[(\lambda x. \mathcal{N}''(\mathfrak{p})) \llbracket N_2 \rrbracket_{\mathfrak{p}''}]$$

and the result follows by using rules INABS, APP1, NINABS, and NAPP1 and induction.

* Assume $N_1 = \mathbf{com}_{\mathfrak{q}, \mathfrak{p}}$. Then if $\mathfrak{q} \neq \mathfrak{p}$, $\llbracket M \rrbracket_{\mathfrak{q}} = \mathbf{send}_{\mathfrak{p}} \llbracket N_2 \rrbracket_{\mathfrak{q}}$, $\llbracket M \rrbracket_{\mathfrak{p}} = \mathbf{recv}_{\mathfrak{p}} \perp$, and for $\mathfrak{p}' \notin \{\mathfrak{q}, \mathfrak{p}\}$, $\llbracket N_1 \rrbracket_{\mathfrak{p}'} = \perp = \llbracket M \rrbracket_{\mathfrak{p}'}$, and therefore $\mathfrak{P} = \mathfrak{q}, \mathfrak{p}$, and if $\mathfrak{q} = \mathfrak{p}$ then $\llbracket N_1 \rrbracket_{\mathfrak{p}} = \lambda x. x$.

If $\mathfrak{P} = \mathfrak{q}, \mathfrak{p}$ then $\mathcal{N} = \llbracket M \rrbracket \setminus \{\mathfrak{q}, \mathfrak{p}\} \mid \mathfrak{q}[\perp] \mid \mathfrak{p}[\llbracket N_2 \rrbracket_{\mathfrak{q}} [\mathfrak{q} := \mathfrak{p}]]$. Because $\llbracket N_2 \rrbracket_{\mathfrak{p}} = \perp$ and $\llbracket N_2 \rrbracket_{\mathfrak{q}} = V$, $N_2 = V$. Therefore $M \xrightarrow{P}_D V[\mathfrak{q} := \mathfrak{p}]$ and the result follows.

If $\mathfrak{P} = \mathfrak{p}$ then $\mathfrak{q} = \mathfrak{p}$, $\mathcal{N} = \llbracket M \rrbracket \setminus \{\mathfrak{p}\} \mid \mathfrak{p}[\llbracket N_2 \rrbracket_{\mathfrak{p}}]$ and the rest is similar to above.

* Assume $N_1 = \mathbf{fst}$. Then $N_2 = \mathbf{Pair} V V'$ and for any process $\mathfrak{p} \in \text{pn}(\text{type}(\mathbf{Pair} V V'))$, $\llbracket M \rrbracket_{\mathfrak{p}} = \mathbf{fst} \mathbf{Pair} \llbracket V \rrbracket_{\mathfrak{p}} \llbracket V' \rrbracket_{\mathfrak{p}}$ and for any other process $\mathfrak{p}' \notin \text{pn}(\text{type}(\mathbf{Pair} V V'))$, by Lemma 38 we have $\llbracket M \rrbracket_{\mathfrak{p}'} = \llbracket N_1 \rrbracket_{\mathfrak{p}'} = \perp$, and therefore $\llbracket M \rrbracket_{\mathfrak{p}'} \rightarrow$.

If $\mathfrak{P} = \mathfrak{p} \in \text{pn}(\text{type}(\mathbf{Pair} V V'))$ then $\mathcal{N} = \llbracket M \rrbracket \setminus \{\mathfrak{p}\} \mid \mathfrak{p}[\llbracket V \rrbracket_{\mathfrak{p}}]$ and $M \xrightarrow{P}_D V$. The result follows by use of rule NPROJ1 and Lemma 38.

* Assume $N_1 = \mathbf{snd}$. This case is similar to the previous.

* Otherwise, $N_1 \neq V$ and either $\llbracket M \rrbracket \xrightarrow{\tau}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$ or $\llbracket M \rrbracket \xrightarrow{\tau, \mathfrak{a}}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$.

If $\llbracket M \rrbracket \xrightarrow{\tau}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$ then either $\llbracket N_1 \rrbracket_{\mathfrak{p}} \xrightarrow{\tau} B$ and $\mathfrak{p} \in \text{pn}(\text{type}(N_1))$, $\mathcal{M} = \llbracket M \rrbracket \setminus \{\mathfrak{p}\} \mid \mathfrak{p}[B \llbracket N_2 \rrbracket_{\mathfrak{p}}]$. We therefore have $\llbracket N_1 \rrbracket \xrightarrow{\tau}_{\mathfrak{p}} \llbracket N_1 \rrbracket \setminus \{\mathfrak{p}\} \mid \mathfrak{p}[B]$, and by induction, $N_1 \rightarrow^*_D N'_1$ such that $\llbracket N_1 \rrbracket \setminus \{\mathfrak{p}\} \mid \mathfrak{p}[B] \rightarrow^* \mathcal{N}'_1 \supseteq \llbracket N'_1 \rrbracket$. Since all these transitions can be propagated past N_2 in the network and $\llbracket N_2 \rrbracket_{\mathfrak{p}'}$ in any process \mathfrak{p}' involved, we get the result for $M' = N'_1 N_2$.

If $\llbracket M \rrbracket \xrightarrow{\tau, \mathfrak{a}}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$ then the case is similar.

– If $N_2 \neq V$ then we have 2 cases.

* If $\llbracket M \rrbracket \xrightarrow{\tau}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$ then either $\llbracket N_1 \rrbracket_{\mathfrak{p}} \xrightarrow{\tau} B$ or $\llbracket N_2 \rrbracket_{\mathfrak{p}} \xrightarrow{\tau} B$ and the case is similar to the previous.

* If $\llbracket M \rrbracket \xrightarrow{\tau, \mathfrak{a}}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$ then there exists L such that either $\llbracket N_1 \rrbracket_{\mathfrak{q}} \xrightarrow{\mathbf{send}_{\mathfrak{p}} L} B_{\mathfrak{q}}$ or $\llbracket N_2 \rrbracket_{\mathfrak{q}} \xrightarrow{\mathbf{send}_{\mathfrak{p}} L} B_{\mathfrak{q}}$ and $\llbracket N_1 \rrbracket_{\mathfrak{p}} \xrightarrow{\mathbf{recv}_{\mathfrak{q}} L[\mathfrak{q} := \mathfrak{p}]} B_{\mathfrak{p}}$ or $\llbracket N_2 \rrbracket_{\mathfrak{p}} \xrightarrow{\mathbf{recv}_{\mathfrak{q}} L[\mathfrak{q} := \mathfrak{p}]} B_{\mathfrak{p}}$.

If $\llbracket N_1 \rrbracket_{\mathfrak{q}} \xrightarrow{\mathbf{send}_{\mathfrak{p}} L} B_{\mathfrak{q}}$ then $\llbracket N_1 \rrbracket_{\mathfrak{q}} \neq L'$ and therefore $\llbracket N_1 \rrbracket_{\mathfrak{p}} \xrightarrow{\mathbf{recv}_{\mathfrak{q}} L[\mathfrak{q} := \mathfrak{p}]} B_{\mathfrak{p}}$ and the

case is similar to the previous. If $\llbracket N_2 \rrbracket_q \xrightarrow{\text{send}_p L} B_q$ then $\llbracket N_1 \rrbracket_q = L'$, and therefore $\llbracket N_2 \rrbracket_p \xrightarrow{\text{recv}_q L[q:=p]} B_p$ and the case is similar to the previous.

- Assume $M = \mathbf{case} N \mathbf{ of} \mathbf{Inl} x \Rightarrow N'; \mathbf{Inr} x' \Rightarrow N''$. Then for any process $p \in \text{pn}(\text{type}(N))$, $\llbracket M \rrbracket_p = \mathbf{case} \llbracket N \rrbracket_p \mathbf{ of} \mathbf{Inl} x \Rightarrow \llbracket N' \rrbracket_p; \mathbf{Inr} x' \Rightarrow \llbracket N'' \rrbracket_p$. And for any other process $p' \notin \text{pn}(\text{type}(N))$, $\llbracket M \rrbracket_{p'} = (\lambda x. \llbracket N' \rrbracket_{p'} \sqcup \llbracket N'' \rrbracket_{p'}) \llbracket N \rrbracket_{p'}$. We know that $\llbracket M \rrbracket \xrightarrow{\tau_p}_{\llbracket D \rrbracket} \mathcal{M} \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}$ and we have three cases.
 - Assume $P = p \in \text{pn}(\text{type}(N))$. Then we have three cases.
 - * Assume $N = \mathbf{Inl} V$. Then $\llbracket N \rrbracket_p = \mathbf{Inl} \llbracket V \rrbracket_p$ and $\mathcal{M} = \llbracket M \rrbracket \setminus \{p\} \mid p[\llbracket N' \rrbracket_p[x := \llbracket V \rrbracket_p]]_p$. We define $M'' = N'$ and the transitions used in $\mathcal{M} \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}$ can be used on M'' . By induction, since $\llbracket N' \rrbracket_{p'} \sqsupseteq \llbracket N' \rrbracket_{p'} \sqcup \llbracket N'' \rrbracket_{p'}$ the result follows from using rules NABSAPP and NCASEL.
 - * Assume $N = \mathbf{Inr} V$. Then the case is similar to the previous.
 - * Otherwise, we use either rule NCASE or rule NCASE2. If we use rule NCASE, we have a transition $\llbracket N \rrbracket_p \xrightarrow{\tau} B$ such that

$$M = \llbracket M \rrbracket \setminus \{p\} \mid p[\mathbf{case} B \mathbf{ of} \mathbf{Inl} x \Rightarrow \llbracket N' \rrbracket_p; \mathbf{Inr} x' \Rightarrow \llbracket N'' \rrbracket_p]$$

and the result follows from induction similar to the last application case.

If we use rule NCASE2 then $\llbracket N' \rrbracket_p \xrightarrow{\tau}_{\llbracket D \rrbracket} B$ and $\llbracket N'' \rrbracket_p \xrightarrow{\tau}_{\llbracket D \rrbracket} B$. If $\llbracket N' \rrbracket_p \xrightarrow{\tau}_{\llbracket D \rrbracket} B$ then by induction, $N' \rightarrow_{\llbracket D \rrbracket}^* N'''$ and $\llbracket N' \rrbracket \setminus \{p\} \mid p[B] \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}''$ such that $\mathcal{N}'' \sqsupseteq \llbracket N''' \rrbracket$ and $N'' \rightarrow_{\llbracket D \rrbracket}^* N''''$ and $\llbracket N'' \rrbracket \setminus \{p\} \mid p[B] \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}'''$ such that $\mathcal{N}''' \sqsupseteq \llbracket N'''' \rrbracket$. Since N' and N'' are mergeable on other processes, the result follows from using rule INCASE.

- Assume $P = p \notin \text{pn}(\text{type}(N))$. Then we have three cases.
 - * Assume $N = \mathbf{Inl} V$. Then $\llbracket N \rrbracket_p = \perp$ and $\mathcal{M} = \llbracket M \rrbracket \setminus \{p\} \mid p[\llbracket N' \rrbracket_p \sqcup \llbracket N'' \rrbracket_p]$. We define $M' = N'$ and the result follows.
 - * Assume $N = \mathbf{Inr} V$. Then the case is similar to the previous.
 - * Otherwise, $\llbracket N \rrbracket_p \neq L$ and we therefore have $\llbracket N \rrbracket_p \xrightarrow{\tau} B$ and $\mathcal{M} = \llbracket M \rrbracket \setminus \{p\} \mid p[(\lambda x. \llbracket N' \rrbracket_p \sqcup \llbracket N'' \rrbracket_p) B]$. We therefore have $\llbracket N \rrbracket \xrightarrow{\tau_p}_{\llbracket D \rrbracket} \llbracket N \rrbracket \setminus \{p\} \mid p[B]$, and by induction, $N \rightarrow_{\llbracket D \rrbracket} N'''$ such that $\llbracket N \rrbracket \setminus \{p\} \mid p[B] \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}'''$ for $\mathcal{N}''' \sqsupseteq \llbracket N''' \rrbracket$. Since all these transitions can be propagated past N_2 in the network and the conditional or $(\lambda x. \llbracket N' \rrbracket_{p'} \sqcup \llbracket N'' \rrbracket_{p'})$ in any other process p' involved, we get the result for $M' = \mathbf{case} N''' \mathbf{ of} \mathbf{Inl} x \Rightarrow N'; \mathbf{Inr} x' \Rightarrow N''$.
- Assume $P = q, p$. Then the logic is similar to the third subcases of the previous two cases.
- Assume $M = \mathbf{select}_{q,p} \ell N$. This is similar to the $N_1 = \mathbf{com}_{q,p}$ case above.
- Assume $M = f(p_1, \dots, p_n)$. Then

$$\llbracket M \rrbracket = \prod_{i=1}^n p_i[f_i(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n)] \mid \prod_{p \notin \{p_1, \dots, p_n\}} p[\perp]$$

We therefore have some process p such that $P = p$ and $(\llbracket M \rrbracket \setminus p_i \mid p_i[\llbracket D \rrbracket(f_i(\vec{p}))][\vec{p}' := p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n]] \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}$. We then define the required choreography $M'' = D(f(p'_1, \dots, p'_n))[\vec{p}' := p_1, \dots, p_n]$ and network



$$\mathcal{N} = \llbracket M'' \rrbracket = \prod_{i=1}^n p_i[\llbracket D \rrbracket(f_i(\vec{p}))][\vec{p}' := p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n] \mid \prod_{p \notin \{p_1, \dots, p_n\}} p[\perp]$$

and the result follows from induction. ◀

Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

Jan de Muijnck-Hughes  

University of Glasgow, UK

Wim Vanderbauwhede  

University of Glasgow, UK

Abstract

Quantitative Type-Systems support fine-grained reasoning about term usage in our *programming languages*. Hardware Design Languages are another style of language in which quantitative typing would be beneficial. When wiring components together we must ensure that there are no unused ports, dangling wires, or accidental fan-ins and fan-outs. Although many wire usage checks are detectable using static analysis tools, such as Verilator, quantitative typing supports making these extrinsic checks an intrinsic aspect of the type-system. With quantitative typing of bound terms, we can provide design-time checks that all wires and ports have been used, and ensure that all wiring decisions are explicitly made, and are neither implicit nor accidental.

We showcase the use of quantitative types in hardware design languages by detailing how we can retrofit quantitative types onto SystemVerilog netlists, and the impact that such a quantitative type-system has when creating designs. Netlists are gate-level descriptions of hardware that are produced as the result of synthesis, and it is from these netlists that hardware is generated (fabless or fabbed). First, we present a simple structural type-system for a featherweight version of SystemVerilog netlists that demonstrates how we can type netlists using standard structural techniques, and what it means for netlists to be type-safe but still lead to ill-wired designs. We then detail how to retrofit the language with quantitative types, make the type-system sub-structural, and detail how our new type-safety result ensures that wires and ports are used once.

Our ideas have been proven both practically and formally by realising our work in Idris2, through which we can construct a verified language implementation that can type-check existing designs. From this work we can look to promote quantitative typing back up the synthesis chain to a more comprehensive hardware description language; and to help develop new and better hardware description languages with quantitative typing.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Language features; Software and its engineering → Domain specific languages; Software and its engineering → System modeling languages

Keywords and phrases Hardware Design, Linear Types, Dependent Types, DSLs, Idris, SystemVerilog, Netlists

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.8

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.4>

Software (Source Code): <https://github.com/border-patrol/linear-circuits>

archived at [swh:1.dir:7faf6fa5d5aaccec3287a1359a52a16acf1be1f5](https://swh.1.dir:7faf6fa5d5aaccec3287a1359a52a16acf1be1f5)

Funding The work is funded by EPSRC grants: Border Patrol (EP/N028201/1) and AppControl (EP/V000462/1).

1 Introduction

Quantitative Type System (*QTS*) support fine-grained reasoning about term usage in our *programming languages*, such that we can control precisely how many times a bounded term can be used. We are even starting to see *QTSs* being introduced into general purpose programming languages, for example: Linear Haskell [7], Idris2 [10], and Granule [33]. Issues around term usage are, however, not limited to programming languages.



© Jan de Muijnck-Hughes and Wim Vanderbauwhede;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 8; pp. 8:1–8:28



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Widely used Hardware Description Languages (*HDLs*) capture not only how hardware designs behave but also how they are physically structured. SystemVerilog is a leading industry standard *HDL* that supports the modelling, verification, and fabrication of hardware designs [14, 23]. Within SystemVerilog a hardware design consists of a high-level structural description of hardware in which modules represent physical units of design, are connected with wires, and each module contains descriptions of the modules behaviour. These behavioural descriptions describe how values are either initially placed on a wire, modified, or distributed through other channels. From this behavioural description, the design is *verified* (tested) to ensure that it behaves as intended. Verification happens using a mixture of static analysis tooling (to externally analyse the design), and testbench creation (in the same language) that is then simulated. Once the design has been verified, a *synthesis* tool converts the high-level description into a gate-level netlist; depending on the complexity of the described design, synthesis can take hours to complete. Netlists sit at the end of the synthesis process that takes high-level hardware designs and generates a single description from which hardware is then generated for fabrication or deployed in fabless environments. These netlist descriptions are also written in SystemVerilog, and the low-level gates offered can differ per hardware platform.

Term usage within *HDLs* is, unfortunately, different to that seen in general purpose programming languages. Bounded terms represent, quite literally, a physical resource that is to be used. Figure 1 presents several illustrative valid hardware designs as specified in Verilog, a subset of SystemVerilog. Each module has an explicit *header* (also known as a portgroup) that details the module’s interface with the outside world. Portgroups detail the typed ports in the interface and the direction that signals can flow through each port. Within modules, we can connect ports to logic gates (or other synthesisable terms), introduce internal wiring, and make direct assignments between ports. Now let us focus on the wiring in our illustrative examples: Figure 1a presents a valid repeated use of a wire; Figure 1b presents a potentially invalid fan-in in which the two inputting wires are negated and wired into the outputting port; Figure 1c presents bit-vector indexing that requires repeated term usage; and Figure 1d presents use of an intermediate variable to reroute part of a bit-vector, and that the bit-vector’s last port is not used. All these examples are structurally valid designs yet the wiring decisions (if a wire is to be used or not, and how often it is to be used) are not always explicit. Existing tools, such as *Verilator*¹, support *external* static analyses of hardware designs that includes reasoning about wire usage. Such static analyses can detect, amongst other things, if wires are unused, and if outputting ports are driven by multiple wires. If such wiring issues are not resolved they could physically damage the hardware and produce ill-behaved hardware.

An issue arising from using external tooling is that their checks are external to the design being captured, thus allowing a *time-of-check time-of-use* issue to exist between when a design is checked and when a design is used. With the rising popularity of *QTS* we must ask ourselves: Can quantitative reasoning about term usage also be applied to *HDLs*? With such quantitative reasoning we can start to embed wire usage constraints directly into our *HDL*’s type-system, such that our external checks (using external tools) now become an intrinsic aspect of the language design itself. By using a more expressive quantitative type-system we can ensure that any and all wiring decisions are explicit, and that ill-wired designs cannot be expressed as they are ruled out by the type-system as ill-typed designs.

¹ <https://verilator.org/>

<pre> module Example(output logic c, input logic a); nand(c, a, a); endmodule; </pre> <p>(a) Repeated Wire Use.</p> <pre> module Example(output logic c, input logic[1:0] ab); and(c, ab[0], ab[1]); endmodule; </pre> <p>(c) Bit-Vector Indexing.</p>	<pre> module Example(output logic c, input logic a,b); wire logic temp; assign temp = b; assign temp = a; not(c, temp); endmodule; </pre> <p>(b) Fan-In.</p> <pre> module Example(output logic c, input logic[2:0] ab); wire logic temp; temp = ab[0]; and(c, temp, ab[1]); endmodule; </pre> <p>(d) Intermediate Wiring.</p>
--	---

■ **Figure 1** Example (System)Verilog.

1.1 Contributions

Existing efforts in developing calculi for verifying hardware designs in Verilog-like languages have not sought to reason about quantitative wire usage [28, 31, 26]. Leaning on the idea of *featherweight* languages [21, 36, 34, 24] we have explored a type-based approach to reasoning about linear wire usage for *netlists* using Quantitative Type Theory (*QTT*) [4]. By capturing a valid subset of Verilog we can type existing designs to ensure that all wires are used exactly once, and showcase our approach. Moreover, we now have a foundation for future work that can investigate ways in which the calculi can be extended up the synthesis chain *formally* (inline with the formal specification); and *safely* (such that known properties of the specification still hold).

Specifically, our contributions are:

- We provide a formal unrestricted type-system (*CIRCUITS*) for Verilog netlists, and introduce notions of type-(un)safety based on denotational semantics relating to graph topology.
- We provide a formal restricted type-system (*CIRQTS*) that is resource-oriented, and detail how it provides fine-grained resource tracking for wires and ports without the need for user annotations. Using *CIRQTS* we ensure that wire and ports can only be connected to once and we detail how we can supply new Verilog primitives that support explicit weakening of a wire’s linearity safely. This weakening enables modelling of existing valid designs, and ensure wiring decisions are explicitly made.
- We put theory into practice by building a mechanised verified implementation of *CIRCUITS* and *CIRQTS* in Idris2 such that we can type-check existing valid Verilog netlists (modulo implementation restrictions and technology mapping) to detect wire usage violations. The implementation of *CIRCUITS* and *CIRQTS* has been made freely available online, and can also be found in the accompanying supplementary material.

With *CIRCUITS* and *CIRQTS* we have developed a better understanding of how to: fundamentally type wiring in Verilog; and produce a type-system that makes wiring safer. Moreover, with *CIRQTS* we have also begun to understand what impact linear wire usage has on circuit design. Although we can reason linearly about wire usage, we have found that

sometimes linear usage is too restrictive and dependent on the intended behaviour of the design. It remains an open question over how best we can weaken linearity in more nuanced ways needed for hardware design.

1.2 Outline

Section 2 details a formal abstract syntax for Verilog netlists that our two type-systems type. Our unrestricted netlist language (CIRCUITS) is formally presented in Section 3, and the linearly wired variant (CIRQTS) in Section 4. For both calculi we detail their *wire-safety*, our approach to type-safety as an interpretation to a graph, and how CIRCUITS is wire unsafe but CIRQTS is not. With linear wiring, wire endpoints and ports, cannot be reused. We detail in Section 5 new *easy-to-add* primitives to Verilog (as primitive gates), that support valid weakening of linearity to allow wires to be split and combined. Section 6 details how we have mechanised our realisation of both languages in Idris2, and Section 7 discusses the results of using CIRQTS for designing netlists. From our experimentation we discuss the efficacy of our approach in Section 7. We end by relating our contributions to the wider world in Section 8.

2 A Syntax for Netlists

Figure 2 presents the shared abstract syntax for CIRCUITS and CIRQTS. Both languages abstract over a subset of Verilog’s concrete syntax required for gate-level modelling. The syntax is unremarkable, which highlights the simplicity that accompanies netlist language design.

$$\begin{aligned}
 n : \mathbb{N} &::= \text{Natural Numbers} \\
 d : \mathcal{D} &::= \text{I} \mid \text{O} \mid \text{IO} \\
 i : \mathcal{I} &::= \text{Logic} \mid [i ; n] \\
 g &::= \text{mux2}(e, e, e, e) \mid b(e, e, e), b \in \{\text{and, or, } \dots\} \mid u(e, e), u \in \{\text{not, } \dots\} \\
 e &::= \varphi \mid \text{port } i \text{ d as } \varphi \text{ in } e \mid \text{gate } g \text{ as } \varphi \text{ in } e \mid \text{wire } i \text{ as } \varphi \text{ in } e \mid \text{assign } e \leftarrow e \text{ in } e \\
 &\quad \mid \text{stop} \mid (\text{index } e \ n) \mid (\text{cast } e \ f) \mid (\text{readFrom } e) \mid (\text{writeTo } e)
 \end{aligned}$$

■ **Figure 2** Abstract Syntax for CIRCUITS and CIRQTS.

We restrict our modelling to simple synthesisable datatypes (\mathcal{I}) as seen in Verilog: 4-state logic bits and their aggregation into bit-vectors. The set of supported gates (g) is indicative, as many netlists are dependent on the technology (gates) as supported by the underlying hardware. For our investigation we explored two input multiplexers, binary logic gates, and unary logic gates, as this presents an expressive enough set of gates to create interesting designs. Although we do not consider gates with n-ary outputs, such as demultiplexers, our setting does support them as we shall see in Section 4.2. Whilst this language seems overly restrictive when compared to state-of-the-art *HDLs*, the syntax and type-system can be extended to deal with n-ary gate syntax, and new primitive gates added.

Most notable from Figure 2 is that the top-level module header is implicit, and sequencing (recall that Verilog is imperative) of statements is realised as continuations on desired sub-terms. Circuit designs represent a single module with a predefined set of ports (supporting uni-directional and bi-directional signals), as such we can represent the module header as a series of bespoke let-bindings that introduce bound terms (with variables being represented

by φ) into the corresponding scope. Declarations for gates and internal wires follow this same pattern, as does the direct assignment (connection) of ports and wires which is supported by the assign statement. Verilog also supports anonymous gate declarations, to keep our minimal calculi small we have purposefully not incorporated them into the syntax but note their addition is straightforward. Finally, the `stop` expression indicates the end of the netlist specification.

The final set of expressions are required to capture type-safe wiring of ports and wires to gates. Naturally, bit-vectors can be indexed to access individual wires in the vector. We do not support slicing as it can be elaborated/synthesised into our core syntax. The remaining expressions are however, not explicit in Verilog’s concrete syntax. Nonetheless, they can be inserted automatically through syntax elaboration.

The first of these hidden expressions supports the insertion of bi-directional ports into gates. Gates only have ports that are inputting or outputting, but bound ports may be bi-directional. `Cast`, like `index`, is an in-place operation on ports and endpoints, that supports transformation of the given direction to supplied direction f .

The final two expressions relate to inserting internal wires into declared gates. Verilog’s concrete syntax does not discriminate between ports and wire endpoints. We thus introduce two projection terms to support such access, one that supports reading from a channel, and the other to support writing to a channel.

We end this section with Figure 3 that illustrates how Figure 1d is encoded in the shared syntax.

```
port Logic O as c in
port Logic I as ab in
wire Logic as temp
  in assign (writeTo temp) ← (index ab 0)
  in gate and (c, (readFrom temp), (index ab 1)) as ga
in stop
```

■ **Figure 3** Figure 1d in the shared Syntax.

3 A Structural Type-System

The abstract syntax from Section 2 helps direct well-formed netlists through its syntactical structure. Nonetheless, a type-system will categorically ensure that the netlists are *well-formed*. This section introduces a simple, yet *flawed*, type-system for CIRCUITS in which term usage is unrestricted, but nonetheless respects how netlists are typed. Practically speaking, the flaw allows uncontrolled fan-ins and fan-outs to be inserted unchecked into the design, and for ports and channels to be left dangling (unused). Section 4 details how quantitative wire usage ensures that ports and channels are used exactly once.

3.1 Types and Contexts

Figure 4 details the (unsurprising) set of types and typing context for CIRCUITS. Ports types (\mathcal{P}) are parameterised by how signals flow from ports, together with the port’s shape as captured by the given datatype. Similarly, wire types (\mathcal{W}) are parameterised by the wire’s shape, the provided datatype. Further, gates are given their own type (\mathcal{G}), and the unit type ($\mathbb{1}$) signifies the end of a design.

$$t : \text{TYPE} ::= \mathcal{P}(i, d) \mid \mathcal{W}(i) \mid \mathcal{G} \mid \mathbb{1}$$

$$\Gamma ::= \emptyset \mid \Gamma + x : t$$

■ **Figure 4** Types and Contexts for CIRCUITS.

3.2 Typing Rules

Figure 5 details the typing rules for CIRCUITS. The introduction rules for natural numbers and datatypes are not presented and are implicitly given in Figure 2. Much like the rest of the definition of CIRCUITS, the rules are for the most part unsurprising and follow standard norms. Rule STOP introduces the unit type, signalling the end of the design. Rules MUX, BIN, & UN type gates and, as in Verilog itself, gates present their output ports then their input ports such that all ports have the same type, which for netlists will be: `Logic`. The Rules PORT, WIRE, & GATE adapt the standard presentation for let-bindings in which the body is extended with the introduced variable. Direct assignment (Rule ASSIGN) of an output port to an input port happens if they both have the same shape i.e. datatype. Indexing vector-shaped ports (Rule INDEX) returns a single port from the array with the shape of the contained elements, and the same signal flow. Further, a compile-time bounds check is required to ensure safe vector-indexing.

Gates and direct assignment only support ports with flow input or output. Casting (Rule CAST) supports safe transformation of a bidirectional port to either an input or output, which we formally express as follows:

► **Definition 1** (Valid Casting of Directions). *Given $a, b : \mathcal{D}$ the safe transformation of flow a to b , which we denote as $\text{ValidCast}(a, b)$, is:*

$\text{ValidCast}(a, b)$	I	O	IO
I	×	×	×
O	×	×	×
IO	✓	✓	×

We have purposefully presented a *strict* interpretation of casting, and do not include the *identity* cast. During the mechanised elaboration of terms (Section 6), we wished to ensure that casts are only inserted if the cast would change the direction. Inclusion of the identity cast would not weaken our result.

The final two rules, Rules READ & WRITE detail how a wire's endpoints are accessed by projection. It is here that the typing rules become somewhat counter-intuitive. Intuition tells us that projecting a wire to *read its contents* would return an outputting port, and conversely *writing to a wire* would require its inputting port. Instead we have swapped the directions: reading gives an input; writing gives an output. Such a swap supports the *natural* typing rules when checking gates and direct assignments. That is, by swapping the expected flow of information during projection we do not need to check if a port marked output will go to an input and vice-versa. Checking for direction equality thus simplifies the typing rules, and checking, for CIRCUITS.

$$\begin{array}{c}
\text{VAR} \frac{\varphi : t \in \Gamma}{\Gamma \vdash \varphi : t} \qquad \text{STOP} \frac{}{\Gamma \vdash \text{stop} : \mathbb{1}} \\
\\
\text{MUX} \frac{\Gamma \vdash o : \mathcal{P}(\text{Logic}, \text{O}) \quad \Gamma \vdash c : \mathcal{P}(\text{Logic}, \text{l}) \quad \Gamma \vdash a : \mathcal{P}(\text{Logic}, \text{l}) \quad \Gamma \vdash b : \mathcal{P}(\text{Logic}, \text{l})}{\Gamma \vdash \text{mux2}(o, c, a, b) : \mathcal{G}} \\
\\
\text{BIN} \frac{\Gamma \vdash o : \mathcal{P}(\text{Logic}, \text{O}) \quad \Gamma \vdash x : \mathcal{P}(\text{Logic}, \text{l}) \quad \Gamma \vdash y : \mathcal{P}(\text{Logic}, \text{l}) \quad b \in \{\text{and}, \text{or}, \dots\}}{\Gamma \vdash b(o, x, y) : \mathcal{G}} \\
\\
\text{UN} \frac{\Gamma \vdash o : \mathcal{P}(\text{Logic}, \text{O}) \quad \Gamma \vdash i : \mathcal{P}(\text{Logic}, \text{l}) \quad u \in \{\text{not}, \dots\}}{\Gamma \vdash u(o, i) : \mathcal{G}} \\
\\
\text{PORT} \frac{i : \mathcal{I} \quad d : \mathcal{D} \quad \Gamma + \varphi : \mathcal{P}(i, d) \vdash b : \mathbb{1}}{\Gamma \vdash \text{port } i \text{ d as } \varphi \text{ in } b : \mathbb{1}} \qquad \text{WIRE} \frac{i : \mathcal{I} \quad \Gamma + \varphi : \mathcal{W}(i) \vdash b : \mathbb{1}}{\Gamma \vdash \text{wire } i \text{ as } \varphi \text{ in } b : \mathbb{1}} \\
\\
\text{GATE} \frac{\Gamma \vdash g : \mathcal{G} \quad \Gamma + \varphi : \mathcal{G} \vdash b : \mathbb{1}}{\Gamma \vdash \text{gate } g \text{ as } \varphi \text{ in } b : \mathbb{1}} \\
\\
\text{ASSIGN} \frac{d : \mathcal{I} \quad \Gamma \vdash i : \mathcal{P}(d, \text{l}) \quad \Gamma \vdash o : \mathcal{P}(d, \text{O}) \quad \Gamma \vdash b : \mathbb{1}}{\Gamma \vdash \text{assign } i \leftarrow o \text{ in } b : \mathbb{1}} \\
\\
\text{INDEX} \frac{d : \mathcal{I} \quad f : \mathcal{D} \quad n : \mathbb{N} \quad \Gamma \vdash p : \mathcal{P}([d; m], f) \quad [n < m]}{\Gamma \vdash (\text{index } p \ n) : \mathcal{P}(d, f)} \\
\\
\text{CAST} \frac{d : \mathcal{I} \quad a, b : \mathcal{D} \quad \Gamma \vdash p : \mathcal{P}(d, a) \quad \text{ValidCast}(a, b)}{\Gamma \vdash (\text{cast } p \ a) : \mathcal{P}(d, b)} \\
\\
\text{READ} \frac{d : \mathcal{I} \quad \Gamma \vdash c : \mathcal{W}(d)}{\Gamma \vdash (\text{readFrom } c) : \mathcal{P}(d, \text{l})} \qquad \text{WRITE} \frac{d : \mathcal{I} \quad \Gamma \vdash c : \mathcal{W}(d)}{\Gamma \vdash (\text{writeTo } c) : \mathcal{P}(d, \text{O})}
\end{array}$$

■ **Figure 5** Simple Typing Rules for CIRCUITS.

3.3 Wire-(Un)Safety

As programming languages are computational it makes sense to reason about their type-safety using their operational semantics: How they compute! Standard syntactic approaches require that we describe how terms reduce (progress) during operation, and prove that once a value is reached that the types have been preserved (preservation). *HDLs* on the other hand, not only describe how signals flow across wires, but also the structure of the circuit itself. At the level of netlists, *HDLs* are not computational languages and there is no reduction of terms. We must, therefore be concerned with a design's physical structure rather than its behaviour. As such we will reason about our type system's correctness based on its *wire-safety*, that is how well the design has been wired together, as opposed to how the design behaves. Specifically, we will use a denotational approach in which we use the circuit design as the instructions to

construct a graph, and assess the design’s wire-safety using the constructed graph’s topology. We stress that these results provide assurances over how circuits are wired and not how they behave. Behavioural safety requires more work [22, 28].

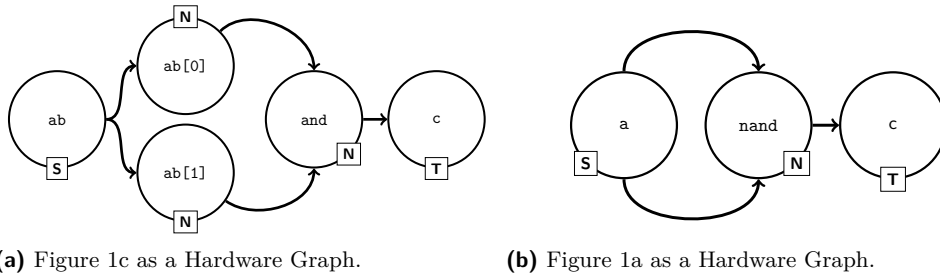
We remark that our denotational approach to wire-safety is inspired by the use of definitional interpreters [38, 37] to prove type-soundness [3]. We elide technical details, highlighting the main results only, and remark that the accompanying artefact (which provides a mechanisation of our contributions, see Section 6) contains the full details.

As a reminder, the “safety” result we present here is purposefully unsafe to capture how a structural type-system does not provide guarantees over the sub-structural property that is wire-safety. The next section Section 4 describes a more (type) safe way to determine wire-safety.

3.3.1 Graphs for Hardware

Digital circuits (netlists) are *just* multigraphs. Leaf vertices map to a netlist’s inputting and outputting ports, and internal vertices represent logic gates. Wires too can be represented by pairs of internal vertices (one for each endpoint, with a single edge to connect the two), and edges in our graph model wiring between ports, channels, and gates. To better reason about our graph’s topology we categorise leaf vertices as being sources (S ; output ports), targets (T ; input ports), or unknown (U ; bidirectional ports). Internal nodes (N) capture gates, casting, and projection of wires. Figure 6 shows how Figures 1a and 1c can be viewed as a hardware graph, and that the edges represent the wiring between ports, gates, and wires. More formally we can describe such graphs as:

► **Definition 2** (Hardware Graph). *Let $H = \langle vs, es \rangle$ be a directed multigraph where vs is a set of vertices, and es is a list of edges. Vertices in H will be labelled with an element of $\{S, T, U, N\}$.*



■ **Figure 6** Example Hardware Graphs for CIRCUITS.

To reason about wire usage we need to define what it means for our hardware graph to be wired. We can determine the expected degree a vertex has directly from its categorisation: source vertices will have zero incoming edges, and zero or more outgoing edges; leaf vertices will have zero or more incoming edges, and zero outgoing edges; bi-directional vertices will have zero or more incoming or outgoing edges; and internal vertices will have zero or more incoming or outgoing edges. Using $\text{deg}^+(v, es)$ and $\text{deg}^-(v, es)$ to calculate the out and in degrees for a vertex v from a given list of edges es , we can compare the given degrees with the expected degrees. If the relations hold then the circuit has been wired. We formally present this definition as:

► **Definition 3** (A Wired Hardware Graph). *Let $H = \langle vs, es \rangle$ be our hardware graph, the wiring of H is well-formed, $C(H)$, if:*

$$\forall v \in vs \begin{cases} S & \text{deg}^+(S, es) \geq 0 \wedge \text{deg}^-(S, es) \equiv 0 \\ T & \text{deg}^+(T, es) \equiv 0 \wedge \text{deg}^-(T, es) \geq 0 \\ U & \text{deg}^+(U, es) \geq 0 \vee \text{deg}^-(U, es) \geq 0 \\ N & \text{deg}^+(N, es) \geq 0 \wedge \text{deg}^-(N, es) \geq 0 \end{cases}$$

Interpretation of terms will result in either: an error related to binding; a hardware graph; a vertex; the empty value (\perp) of the empty type. Port definitions are interpreted into leaf vertices following the port's direction, whereas gates, wires, and casts interpret into internal nodes. The end of a specification is interpreted into the empty type. Port usages, and channel projection, inserts edges into the graph. We can also strengthen the interpretation by making it type-directed (represented by $\llbracket t \rrbracket_T$), where terms from CIRCUITS are mapped to hardware graph concepts.

Formally we can denote the act of interpretation as follows:

► **Definition 4** (Interpretation). *Let Σ_i be an interpretation environment and $H_i = \langle vs, es \rangle$ be a Hardware Graph. We denote the interpretation of CIRCUITS specification $(\Gamma \vdash e : t)$ as:*

$$(\Sigma, H_i) \llbracket e \rrbracket ::= \text{Error} \mid \text{Done}(H_o, v : \llbracket t \rrbracket_T)$$

where H_o is the resulting hardware graph.

The interpretation environment (Σ) stores the result of interpretation for bound terms and is passed around as an input to interpretation explicitly. The accumulator graph (H) carries the resulting graph as we traverse sub-terms. Here subscripts i and o denote an inputting and resulting value. We cannot *just* return a complete graph for each interpretation step as not all terms return a graph. Take ports, for example, they return a vertex when interpreted during the construction of edges, and `stop` returns the empty value.

More information about the interpretation can be found in the accompanying artefact which contains the mechanisation (Section 6) of CIRCUITS.

3.3.2 Well-Typed Circuits are Valid Hardware Graphs

With this high-level description of interpretation we can detail our strong interpretation soundness result, which is our wire-safety result.

► **Theorem 5** (Strong Interpretation Soundness).

$$\frac{\emptyset \vdash e : \mathbb{1} \quad \llbracket e \rrbracket = \text{Done}(\Sigma, H, v)}{\Sigma \equiv \emptyset \quad v \equiv \perp \quad C(H)}$$

Strong interpretation soundness states that interpretation of a closed specification will result in a valid wired hardware graph. A weak soundness result would be that we can construct *just* a valid hardware graph.

3.3.3 Proof Sketch

Our soundness proof needs to ensure that our interpretation will result in a valid hardware graph. We can structure our proof using Siek's "three easy lemmas" [41] but adapted for interpretation as seen with existing work [3] and adapted to build hardware graphs. Siek's

8:10 Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

approach requires that we divide the proof into three sub-lemmas that, when combined, cover all aspects of the interpretation process: 1) interpreting primitives; 2) ensuring that variables are well-scoped; and 3) that doing interpretation will produce a value.

The first lemma deals with typing primitives.

► **Lemma 6 (Primitives).** *Well-typed primitives are not stuck and will interpret correctly.*

Proof. By induction on typing derivations. ◀

The only primitive term in `CIRCUITS` is `stop`, the rest contain interpretable sub-terms. The term `stop` will always interpret.

The next lemma addresses environment lookups. As in existing work [3] we must first define a consistency relation (\vDash) between typing contexts and interpretation environments. Such a relation ensures that as the typing context grows, so will the interpretation environment.

► **Definition 7 (Consistent Interpretation Environments).**

$$\frac{\text{EMPTYENV}}{\emptyset \vDash \emptyset} \qquad \frac{\text{EXTENDENV} \quad \Gamma \vDash \Sigma}{\Gamma + \varphi : t \vDash \Sigma + (\varphi, v : \llbracket t \rrbracket_T)}$$

Using our consistency relation (\vDash) we can describe well-scoped environment lookups. When given an interpretation environment that is consistent with a typing context, we can map bound variables within the typing context to their equivalent bindings in the corresponding interpretation environment. Thus when given a well-scoped variable $\varphi \in \Gamma$ such that φ has type t , then there is a value $(v : \llbracket t \rrbracket_T)$ in the interpretation environment bound to φ .

► **Lemma 8 (Lookup).** *Well-typed interpretation environment lookups are not stuck and will interpret correctly.*

$$\frac{\Gamma \vDash \Sigma \quad \varphi : t \in \Gamma \quad x \equiv \llbracket t \rrbracket_T}{(\varphi, v) \in \Sigma \quad v : x}$$

Proof. By structural induction over the interpretation environment. ◀

As environment lookups are guided by the typing-context, interpretation will succeed because all references to bound terms exist. Otherwise lookup will fail.

The final lemma, which we provide here, details interpretation.

► **Lemma 9 (Interpretation).** *If the interpretation returns a result then the result is a valid hardware graph.*

$$\frac{\Gamma \vdash e : t \quad \Gamma \vDash \Sigma \quad (\Sigma, H_i) \llbracket e \rrbracket ::= (\text{Done } res)}{res = (H_o, v) \quad v : t' \quad t' \equiv \llbracket t \rrbracket_T}$$

Proof. By induction on typing derivations. ◀

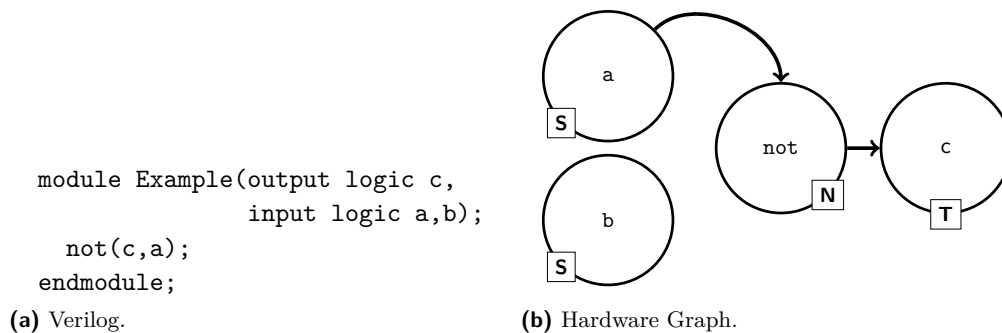
We know that within definitional interpreters the strong soundness property [3] corresponds to the classic strong soundness property used in proving type soundness [47]. With our final lemma we can show that well-typed `CIRCUITS` specifications *will* construct correctly wired hardware graphs, if not an error will occur. Interpretation will finish when the term `stop` is reached. The consistency relation ensures that the typing contexts and interpretation environment remain consistent as we step across binders.

We only check the strong soundness result after the interpretation process has finished. Sub-terms return components of a hardware graph, and not a complete graph itself, we need to ensure that the soundness result is applied to a whole specification.

3.4 Towards True Wire-Safety

Our definition of a wire-safety, and thus safety of `CIRCUITS` type-system, is not one that leads to designs that are inherently *safely wired*: Designs in which wires are used once, and wire usage is clear. The type-system for `CIRCUITS` is not sub-structural, and will admit designs in which ports (and wires) are left dangling or partially used. We know this because our definition for wiredness ($C(H)$) is too loosely specified. We can show this by changing the definition of $C(H)$ to one where the expected degree of leaf nodes is *greater than zero to equal to one*. Such a change now invalidates various graphs in which ports and wires were left dangling, and were once admitted by `CIRCUITS` type-system.

Consider, for example, the netlist and its corresponding hardware graph in Figure 7. The netlist has a dangling input `b`. It is unclear from the immediate context if `b` is supposed to be dangling or included in the design; gates in Verilog are n-ary. The change in definition of $C(H)$ supports identification of ill-wired designs, as the expected out degree of for `b` is one but its given degree is zero.



■ **Figure 7** Example Netlist with Ambiguous Wiring.

Changing the definition of $C(H)$ is still, however, not enough to get safe designs. We need to reason about bidirectional ports, remember that bit-vectors can be indexed, ensure that channel endpoints are accessed once, and that gates are only wired into once. We need a sub-structural type-system to reason about wire/port usage. We will show this in the next section, and provide a better definition of what it means for a hardware graph to be *well-wired*.

4 A Sub-Structural Type-System

The illustrative designs from Figure 1 are not linear in their bound term usage. There are repeated variables, dangling wires, and parts of bit-vectors left unused. Trying to fit *wire usage constraints* into existing quantitative systems is hard. Generally speaking, *QTSs* such as Atkey’s *QTT* [4], and those based around linear logic [45, 44] and graded semirings [33], are designed to reason about term usage within general purpose programming languages. Specifically, linear systems require that bounded terms are used *exactly once*; affine systems require that bounded terms are used *at most once*; graded systems require that bounded terms are used *at most n-times*; and *QTT* allows bounded terms to be used linearly, and unbounded terms have unrestricted usage. *None-One-Tonne* systems do not have a fine grained usage modality. *HDLs* capture circuit behaviour as well as structure.

A key design challenge when *linearising* `CIRCUITS` type-system is to know what *usage* means for *HDLs*, as bounded terms do not have singular usage. Within `CIRCUITS` wires have two endpoints that are used in separate locations, and bit-vector shaped wires/ports can

be indexed in strange and wonderful ways. Roughly, projection of endpoints and indexing implies that each sub-term may be partially used in a design and type-checking needs to take this into account. Further, while ports/wire have bounded usage, gates are unbounded. It is not clear how *QTT* and Granule can be co-opted to encode these domain specific usages as part of a semiring structure. Our approach to the type-system for CIRQTS must be different, for now.

4.1 Types and Usages

Figure 8 details the additional infrastructure required to make CIRCUITS linear. Inspiration is taken from *QTT* to base our system of usages (Figure 8a) on the *none-one-tonne* semiring. Things are free until used, and some things will always be free to use i.e. unbounded usage. In practice, however, this can be distilled to the ordinary boolean semiring as our usage requirements are more conservative and constrained, and we make no link between compile time and runtime quantities as *QTT* is used in practice [10].

$$R ::= 0 \mid 1 \mid \omega \quad \begin{array}{l} i : \mathcal{I} \quad ::= \text{Logic} \\ u : \text{USAGE}_D(i) \quad ::= R \\ \text{Init}_D(u) \quad ::= 1 \end{array} \left| \begin{array}{l} [i; n] \\ \{u_j : \text{USAGE}_D(i)^{j \in 1..n}\} \\ \{\text{Init}_D(u_j)^{j \in 1..n}\} \end{array} \right.$$

(a) Usages.

(b) DataTypes.

$$\begin{array}{l} t : \text{TYPE} \quad ::= \mathcal{P}(i, d) \\ u : \text{USAGE}_T(t) \quad ::= \text{USAGE}_D(i) \\ \text{Init}_T(u) \quad ::= \text{Init}_D(i) \end{array} \left| \begin{array}{l} \mathcal{W}(i) \\ (\text{USAGE}_D(i), \text{USAGE}_D(i)) \\ (\text{Init}_D(i), \text{Init}_D(i)) \end{array} \right| \left. \begin{array}{l} \mathcal{G} \mid \mathbb{1} \\ \omega \mid \omega \\ \omega \mid \omega \end{array} \right.$$

(c) Types.

$$\Gamma ::= \emptyset \mid \Gamma + (\varphi : t, u : \text{USAGE}_T(t)) \mid \Gamma \pm (\varphi : t, u : \text{USAGE}_T(t)) \quad \Gamma_i \vdash e : t \dashv \Gamma_o$$

(d) Contexts.

(e) Judgements.

■ **Figure 8** Types and Contexts for CIRQTS.

Figure 8b presents how we capture datatype usage. Recall that the types for ports and channels are indexed by a datatype. The shape of a port/wire's datatype will direct the usage we need to track. Logic-shaped ports and channels will have a single wire to use, and bit-vectors an array of elements to use. Bit-vectors are, however, multidimensional and the usages should reflect that when indexing bit-vectors. When initialising usages for datatypes we set them to be free.

Figure 8c presents how we specify usages for bindable types. Ports will have to keep track of their usage based on the usage of the datatype they are indexed by, and wires will have a pair of usages (one for each endpoint). Gates (and unit) will be left unrestricted as their use is unrestricted. The syntax for CIRCUITS/CIRQTS enforces that **stop** can only be used once. For newly introduced ports and wires, their initial usage will be free.

Figure 8d details how we situate our usages in the type-system. Following existing work [46] we must annotate our typing context to keep track of the usage of bounded terms. Given the different shapes of our boundable types, we cannot use linear algebra to capture usage updates. Instead we take a simpler approach by envisaging our types (and type-system)

as a stateful resource in which the usage of bound term is a state. Specifically not only can our context be extended, but we can also update the state of a bound term's resource: its usage.

We differ from standard linear typing approaches by not relying on context splitting. Rather our judgement formation (Figure 8e), and typing rules, are more algorithmic. Successfully typing a term will may result in a new updated context. This mirrors the algorithmic presentation of the Linear Lambda Calculus (*LLC*) [45, Figure 1-6].

4.2 New Syntax for Indexing

$$\begin{aligned} (\text{index } e \ n) &\rightarrow (\text{index } \varphi_p \ \{n_i^{i \in 1..m}\}) \\ (\text{readFrom } e) &\rightarrow (\text{readFrom } \varphi_c) \mid (\text{readFromAt } \varphi_c \ \text{id}x) \\ (\text{writeTo } e) &\rightarrow (\text{writeTo } \varphi_c) \mid (\text{writeToAt } \varphi_c \ \text{id}x) \end{aligned}$$

■ **Figure 9** Syntax Changes required for CIRQTS.

We can now start to describe the typing rules for CIRQTS as presented in Figure 10. We must first realise, however, that the abstract syntax (Figure 2) is too expressive. The trouble stems from n-dimensional indexing of bit-vectors. Type-checking each (sub)term in our stateful typing updates the states held within the typing context. How do we know which sub-usage of the bit-vector for which bound term in the typing-context needs to be updated? Indexing a one-dimensional bit-vector is simple, but for deeper indexing (i.e. indexing a projection or an index of an index) updating the state is impossible. We need the location within the context. Thus, we must flatten nested indices to a single term, and have separate terms for when wire endpoints are also projected. This transformation step occurs during elaboration, and Figure 9 presents the replacement terms required. How we can adapt the type-system to support nested indexing is not clear.

4.3 Typing Rules

Figure 10 presents the salient typing rules for CIRQTS. Missing are the rules for the *write* projection as they mirror the rules for the *read* projection. The presented rules are not too dissimilar from those in Figure 5. For each rule we reason, using custom predicates, about the state of the usage resource for each binder, or expect context transitions to enforce our linear wiring property.

Rule VAR ensures that bound variables are only used *if* they are completely free; ruling out use of partially used variables that have been used through indexing. Such a predicate means that once a port/wire has been indexed it can only be used through indexing.

Rule STOP describes the end conditions for each bound term's usage, as dictated by the $\text{CanStop}(t, u)$ predicate that requires that all bound terms must be totally used. That is all ports and wires must be used for the design to type-check. How we define $\text{CanStop}(t, u)$ impacts on what it means for a design to be *totally used*. Verilator, for example, does not require output ports to be fully used in its static analysis. That is, Verilator is affine for outputs but linear for inputs, whereas CIRQTS is linear for all wires/ports. We stress that the termination conditions for CIRQTS are not closed for debate. Rather it supports discussing the conditions under which ports and wires are said to be sufficiently used. Verilator's termination choice, for example, can be replicated by requiring that $\text{CanStop}(t, u)$ requires inputting ports and wire endpoints be used.

8:14 Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

$$\begin{array}{c}
\text{VAR} \frac{(x:t, o) \in \Gamma \quad [\text{IsFree}(o)]}{\Gamma \vdash x:t \vdash \Gamma \pm (x:t, \text{Use}(o))} \quad \text{STOP} \frac{\{\text{CanStop}(t, u) \mid (x:t, u) \in \Gamma\}}{\Gamma \vdash \text{stop} : \mathbb{1} \dashv \emptyset} \\
\text{CAST} \frac{d:\mathcal{I} \quad a, b:\mathcal{D} \quad \Gamma_1 \vdash p:\mathcal{P}(d, a) \dashv \Gamma_2 \quad \text{ValidCast}(a, b)}{\Gamma_1 \vdash (\text{cast } p \ a) : \mathcal{P}(d, b) \dashv \Gamma_2} \\
\text{MUX} \frac{\Gamma_2 \vdash c:\mathcal{P}(\text{Logic}, \mathbb{1}) \dashv \Gamma_3 \quad \Gamma_1 \vdash o:\mathcal{P}(\text{Logic}, \mathbb{O}) \dashv \Gamma_2 \quad \Gamma_3 \vdash a:\mathcal{P}(\text{Logic}, \mathbb{1}) \dashv \Gamma_4 \quad \Gamma_4 \vdash b:\mathcal{P}(\text{Logic}, \mathbb{1}) \dashv \Gamma_5}{\Gamma_1 \vdash \text{mux2}(o, c, a, b) : \mathcal{G} \dashv \Gamma_5} \\
\text{BIN} \frac{\Gamma_1 \vdash o:\mathcal{P}(\text{Logic}, \mathbb{O}) \dashv \Gamma_2 \quad \Gamma_2 \vdash x:\mathcal{P}(\text{Logic}, \mathbb{1}) \dashv \Gamma_3 \quad \Gamma_3 \vdash y:\mathcal{P}(\text{Logic}, \mathbb{1}) \dashv \Gamma_4 \quad b \in \{\text{and}, \text{or}, \dots\}}{\Gamma_1 \vdash b(o, x, y) : \mathcal{G} \dashv \Gamma_4} \\
\text{UN} \frac{\Gamma_1 \vdash o:\mathcal{P}(\text{Logic}, \mathbb{O}) \dashv \Gamma_2 \quad \Gamma_2 \vdash i:\mathcal{P}(\text{Logic}, \mathbb{1}) \dashv \Gamma_3 \quad u \in \{\text{not}, \dots\}}{\Gamma_1 \vdash u(o, i) : \mathcal{G} \dashv \Gamma_3} \\
\text{ASSIGN} \frac{d:\mathcal{I} \quad \Gamma_1 \vdash i:\mathcal{P}(d, \mathbb{1}) \dashv \Gamma_2 \quad \Gamma_2 \vdash o:\mathcal{P}(d, \mathbb{O}) \dashv \Gamma_3 \quad \Gamma_3 \vdash b:\mathbb{1} \dashv \emptyset}{\Gamma_1 \vdash \text{assign } i \leftarrow o \text{ in } b : \mathbb{1} \dashv \emptyset} \\
\text{PORTS} \frac{i:\mathcal{I} \quad d:\mathcal{D} \quad \Gamma + (\varphi:\mathcal{P}(i, d), \text{Init}_T(\mathcal{P}(i, d))) \vdash b:\mathbb{1} \dashv \emptyset}{\Gamma \vdash \text{port } i \ d \text{ as } \varphi \text{ in } b : \mathbb{1} \dashv \emptyset} \\
\text{WIRE} \frac{i:\mathcal{I} \quad \Gamma + (\varphi:\mathcal{W}(i), \text{Init}_T(\mathcal{W}(i))) \vdash b:\mathbb{1} \dashv \emptyset}{\Gamma \vdash \text{wire } i \text{ as } \varphi \text{ in } b : \mathbb{1} \dashv \emptyset} \\
\text{GATE} \frac{\Gamma_1 \vdash g:\mathcal{G} \dashv \Gamma_2 \quad \Gamma_2 + (\varphi:\mathcal{G}, \text{Init}_T(\mathcal{G})) \vdash b:\mathbb{1} \dashv \emptyset}{\Gamma_1 \vdash \text{gate } g \text{ as } \varphi \text{ in } b : \mathbb{1} \dashv \emptyset} \\
\text{INDEX} \frac{d:\mathcal{I} \quad f:\mathcal{D} \quad \text{idx} := \{m_i^{i \in 1..j}\} \quad (\varphi_p:\mathcal{P}([d;n], f), u) \in \Gamma \quad [\text{IsFreeAt}(u, \text{idx})]}{\Gamma \vdash (\text{index } \varphi_p \ \text{idx}) : \mathcal{P}(\text{HasTypeAt}([d;n], \text{idx}), f) \dashv \Gamma \pm (\varphi_p:\mathcal{P}([d;n], f), \text{UseAt}(u, \text{idx}))} \\
\text{READ} \frac{d:\mathcal{I} \quad (\varphi_c:\mathcal{W}(i), (u_r, u_w)) \in \Gamma \quad [\text{IsFree}(u_r)]}{\Gamma \vdash (\text{readFrom } \varphi_c) : \mathcal{P}(d, \mathbb{1}) \dashv \Gamma \pm (\varphi_c:\mathcal{W}(i), (\text{Use}(u_r), u_w))} \\
\text{READAT} \frac{d:\mathcal{I} \quad \text{idx} := \{m_i^{i \in 1..j}\} \quad (\varphi_c:\mathcal{W}(i), (u_r, u_w)) \in \Gamma \quad [\text{IsFreeAt}(u_r, \text{idx})]}{\Gamma \vdash (\text{readFromAt } \varphi_c \ \text{idx}) : \mathcal{P}(d, \mathbb{1}) \dashv \Gamma \pm (\varphi_c:\mathcal{W}([d;n]), (\text{UseAt}(u_r, \text{idx}), u_w))}
\end{array}$$

■ **Figure 10** Typing Rules for CIRQTS.

The rules for gates (Rules MUX, BIN, & UN), casting (Rule CAST), and assignment (Rule ASSIGN) shows how the linearity checking can be propagated *silently* through the type-system. Key to Rules CAST, MUX, BIN, & UN operation is that the premises for ports updates the stateful typing-context. A port can only be used if it is either a variable (Rule VAR) or is the result of indexing a port, or a projection of a wire. Threading the updated state left-to-right along the typing rules ensures that ports cannot be reused, as each sub-term uses the latest version of the context possible.

Terms that introduce binders (Rules PORTS, WIRES, & GATE) are not complicated, and extend the typing-context with a new variable binding with a default usage state.

The final rules deal with indexing ports, and channel projection. Rule INDEX becomes a variant of Rule VAR in which the typing conditions require that instead of the referenced port being completely free, the port must be free at the specified index. Importantly, the return type in Rule INDEX is not the inner type of φ_p but is, instead, the type of element at the end of the presented index. The rules for projection also follow on from Rule VAR in that they resolve references. Whereas Rule READ acts on a port entirely, Rule READAT adapts the structure for Rule INDEX but the usage resource associated with the wire's input endpoint is analysed/updated. Writing to a wire (rules not shown) mirrors those for reading but affect the other resource.

4.4 Wire-Safety

Our approach to reasoning about wire-safety in CIRQTS does not differ much from Section 3.3. The core differences relate to how Hardware Graphs are defined, and what it means for a Hardware Graph to be well wired. We highlight the key differences in approach.

To better reason about our graph's topology we extend the definition of Hardware Graph from Definition 2 and label each vertex with its expected in/out degree. More formally:

► **Definition 10** (Hardware Graph). *Let $H = \langle vs, es \rangle$ be a directed multigraph graph where vs is a partitioned set of vertices, and es a list of edges. Vertices in H are labelled with their minimum in/out degree, and are defined as $v^{o,i}$ where: o is the expected out-degree; i the expected in-degree; and the shape of each v is taken from $\{S^{(n,0)}, T^{(0,n)}, U^{(n,n)}, N(n,m)^{(n,m)}\}$.*

Like the previous definition of a hardware graph we must define what it means for such a graph to be wired. Again, we do so by checking that the expected degrees for each vertex must match the degrees as calculated from the set of edges. Our previous definition, however, is too permissive. Must refine our definition to ensure that the given degrees will match the expected degrees. Moreover, we must ensure that bi-directional ports are used in one direction only, thus we require an exclusive disjunction ($\underline{\vee}$) between the calculated in/out degrees and the expected degrees. Formally we present his definition as:

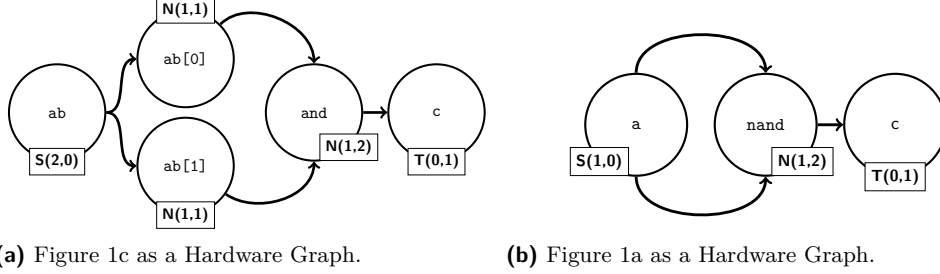
► **Definition 11** (Well-Wired Hardware Graph). *Let $H = \langle vs, es \rangle$ be our hardware graph, the wiring of H is well-formed, $\mathcal{W}(H)$, if $\forall v^{(o,i)} \in vs$:*

$$v^{(o,i)} \left\{ \begin{array}{l} S^{(o,0)} \quad \text{deg}^+(S^{(o,0)}, es) \equiv o \wedge \text{deg}^-(S^{(o,0)}, es) \equiv 0 \\ T^{(0,i)} \quad \text{deg}^+(T^{(0,i)}, es) \equiv 0 \wedge \text{deg}^-(T^{(0,i)}, es) \equiv i \\ U^{(o,i)} \quad \text{deg}^+(U^{(o,i)}, es) \equiv o \wedge \text{deg}^-(U^{(o,i)}, es) \equiv 0 \\ \underline{\vee} \\ \text{deg}^+(U^{(o,i)}, es) \equiv 0 \wedge \text{deg}^-(U^{(o,i)}, es) \equiv i \\ N^{(o,i)} \quad \text{deg}^+(N^{(o,i)}, es) \equiv o \wedge \text{deg}^-(N^{(o,i)}, es) \equiv i \end{array} \right.$$

Figure 11 shows how Figures 1a and 1c can be viewed using the amended hardware graph. With the new hardware graph definition correspondences (Figure 11a) and discrepancies (Figure 11b) can be seen between the expected and given degrees for the vertices in each example. This demonstrates the ability of our new hardware graph definition to more accurately reason about wiring. If we compare these graphs with those in Figure 6 we can better see the accuracy. Both Figure 11a and Figure 6a presents graphs in which there are

8:16 Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

no discrepancies. Figure 11b, however, now shows a discrepancy for vertex a in which its expected out degree is one but its given out degree is two, this was not seen in Figure 6b as there is insufficient information to decide.



(a) Figure 1c as a Hardware Graph.

(b) Figure 1a as a Hardware Graph.

■ **Figure 11** Example Hardware Graphs for CIRQTS.

Interpretation of Verliog to hardware graphs is as before, however, to reason more accurately about bit-vector wiring we calculate the degree of a vertex based on the size of its datatype. The shape logic has size one, and bit-vectors have the size of its element multiplied by the size of the bit-vector. We also need to take into account that type-checking will alter the state associated with bound types. Thus, we must return both the accumulated hardware graph (H_o) and the resulting interpretation environment (Σ_o).

► **Definition 12** (Interpretation). *Let Σ_i be an interpretation environment and $H_i = \langle vs, ws \rangle$ be a Hardware Graph. We denote the interpretation of a CIRQTS design ($\Gamma_i \vdash e : t \dashv \Gamma_o$) as:*

$$(\Sigma_i, H_i) \llbracket e \rrbracket ::= \text{Error} \mid \text{Done}(\Sigma_o, H_o, v : \llbracket t \rrbracket_T)$$

where Σ_o and H_o are the updated environment and resulting hardware graph.

With this new interpretation definition we must also update our interpretation soundness result accordingly.

► **Theorem 13** (Strong Interpretation Soundness).

$$\frac{\emptyset \vdash e : \mathbb{1} \quad \llbracket e \rrbracket = \text{Done}(\Sigma, H, v)}{\Sigma \equiv \emptyset \quad v \equiv \perp \quad \mathcal{W}(H)}$$

and also the final lemma which details interpretation.

► **Lemma 14** (Interpretation). *Interpretation of a design is not stuck and will return a result containing a valid hardware graph.*

$$\frac{\Gamma_i \vdash e : t \dashv \Gamma_o \quad \Gamma_i \models \Sigma_i \quad (\Sigma_i, H_i) \llbracket e \rrbracket ::= (\text{Done } res)}{res = (\Sigma_o, H_o, v) \quad \Gamma_o \models \Sigma_o \quad v : t' \quad t' \equiv \llbracket t \rrbracket_T}$$

Note that we also require a consistency relation on outgoing typing contexts and interpretation environments, as well as the inputting ones. This is required, as stepping through each sub-term modifies the type-level state, and returns a new interpretation environment. Further we need to extend the consistency relation to account for context updates.

► **Definition 15** (Consistent Interpretation Environments under Context Updates).

$$\frac{\text{UPDATEENV} \quad \Gamma \models \Sigma}{\Gamma \pm \varphi : t \models \Sigma \pm (\varphi, v : \llbracket t \rrbracket_T)}$$

With these revised definitions we can now reason about the type system of CIRQTS and its linearity guarantees. The proof that well-typed designs are well-wired hardware graphs does not change from the simply-typed proof sketch in Section 3.3.

An interesting aspect of our formulation, is that we can use the same wire-safety result defined here to show that designs with CIRCUITS specification will be unsafe as the resulting hardware graphs are not linearly wired.

Unfortunately, we are still not done. CIRQTS is too restrictive in its approach to linear wiring. Our type-system, as it stands, will rule out valid designs by removing the ability to fan-out and fan-in when it is benign to do so. Verilog designs require that we can split and join wires together i.e. weaken linearity. The next section discusses how we can extend the *syntax* to support this.

5 Weakening Linearity for Free with new Gates

Figure 12a presents a variant of Figure 1a with many potential linear violations, specifically that the use of `a` is non-linear and that `b` is unused. From this example it is, however, unclear what the designer’s intended wiring was. What was accidental: the repeated use of `a`, or specification of `b` as an input? The linearity encoded within the type-system for CIRQTS is too restrictive. There will be valid cases, for example driving n-ary logic gates with the same input (Figures 1a and 12a), in which we need to weaken linearity to support valid designs. At the same time though, we need to report accidental dangling and repeated wiring.

	<pre>module Example0(output logic c, input logic a);</pre>
<pre>module Example0(output logic c, input logic a,b);</pre>	<pre> wire logic temp0, temp1; split(temp0, temp1, a); nand(c, temp0, temp1);</pre>
<pre> nand(c, a, a);</pre>	<pre>endmodule;</pre>
<pre>endmodule;</pre>	<pre>endmodule;</pre>
(a) Unclear usage.	(b) Clear usage.

■ **Figure 12** Example showing unclear usage violation, and new syntax to make usage explicit.

Figure 13 presents two new gate primitives that support wire duplication (splitting), and wire joining (collection). With these new primitives designers must now *explicitly* state when wires are to be duplicated (i.e. split) and when they are merged (i.e. drive the same output). Presenting them as primitives addresses issues of backwards compatibility as the SystemVerilog standard [23] supports new gate primitives to be introduced: No new syntax changes are required! More so, our wire-safety result from Section 4.4 need not change either.

With the addition of these new primitives we can use internal wiring to rewrite Figure 12a as Figure 12b if the duplication was required, and if not replace an `a` with the `b`. Again we stress that the implicit wiring decision (duplication) is now an explicit one. Figure 14 illustrate this further by comparing the resulting hardware graph and circuit diagrams for Figure 12b.

6 Mechanisation and Realisation in a Dependently-Typed Language

We have mechanised the implementations of CIRCUITS and CIRQTS using Idris2 [10, 9], a general purpose dependently-typed language. Leveraging Idris2’s support for dependent types we can provide both a verifiable *practical* type-checker for each language, but also formally

8:18 Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

$e ::= \dots \mid \text{collect}(e, e, e) \mid \text{split}(e, e, e)$

(a) Syntax.

$$\text{COLLECT} \frac{\Gamma_1 \vdash o : \mathcal{P}(\text{Logic}, O) \dashv \Gamma_2 \quad \Gamma_2 \vdash i_a : \mathcal{P}(\text{Logic}, I) \dashv \Gamma_3 \quad \Gamma_3 \vdash i_b : \mathcal{P}(\text{Logic}, I) \dashv \Gamma_4}{\Gamma_1 \vdash \text{collect}(o, i_a, i_b) : \mathcal{G} \dashv \Gamma_4}$$

$$\text{SPLIT} \frac{\Gamma_1 \vdash o_a : \mathcal{P}(\text{Logic}, O) \dashv \Gamma_2 \quad \Gamma_2 \vdash o_b : \mathcal{P}(\text{Logic}, O) \dashv \Gamma_3 \quad \Gamma_3 \vdash i : \mathcal{P}(\text{Logic}, I) \dashv \Gamma_4}{\Gamma_1 \vdash \text{split}(o_a, o_b, i) : \mathcal{G} \dashv \Gamma_4}$$

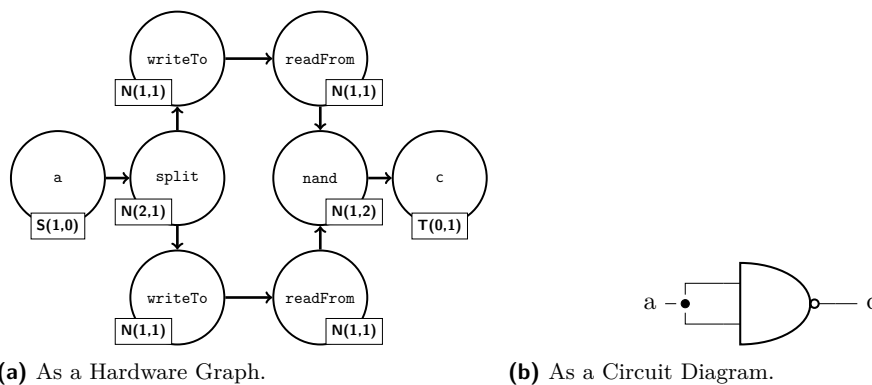
(b) Typing Rules.

```
primitive collect ( output logic o, input logic a, b);
  assign o = a;
  assign o = b;
endprimitive;
```

```
primitive split ( output logic a, b, input logic i);
  assign a = i;
  assign b = i;
endprimitive;
```

(c) Verilog Primitives.

■ **Figure 13** New Primitive Gates to Safely “Weaken” Linearity.



■ **Figure 14** Figure 12b as a Hardware Graph and Circuit Diagram.

reason about wire-safety. Thus, our implementations primarily support the type-checking of netlists (modulo syntax restrictions from Section 2) written in Verilog. With CIRCUITS we can check if it is valid Verilog, with CIRQTS we can check if the netlists are linearly wired. Further, for both type-checkers we incorporated a soundness check (modelled on the type-safety result from CIRQTS) to show which designs accepted by CIRCUITS are in-fact *wire unsafe*.

Both CIRCUITS and CIRQTS have been realised as intrinsically well-typed well-scoped Embedded Domain Specification Languages (*EDSLs*) within Idris2 using well documented techniques [27, 2, 29, 25]. The terms, types, and usages for our languages translate directly into standard dependently (and non-dependently) typed data structures. Reading of valid Verilog netlists comes from a frontend Domain Specification Language (*DSL*) coupled with an elaborator (also known as the type, scope, and usage checker) to construct intrinsically typed terms.

There are three interesting aspects with our approach to the mechanisation: relation between soundness check and our formal proof; intrinsic linearity checking; and error reporting.

Existing work has demonstrated how *well-typed* definitional interpreters can be realised within dependently-typed languages [40, 35] to provide a mechanised runtime. We borrow this approach to not only reason about our type-safety result (wire-safety) as code (i.e. its mechanisation) but make the proof an integral aspect of our tool’s operation.

The core *EDSLs* representing CIRQTS are intrinsically typed both structurally and sub-structurally. Our approach is inspired by *Leftover Typing* [1] in which variable usage updates a type-level state iff a variable is “free” to be used. We use a bespoke list quantifier (presented in Figure 15) paired with a generic update datatype and decision procedure (Figure 16) to provide a type-safe updating of the typing context respective to the predicate being asserted. The type-level constraints that act on variables, that enable our linearity, are instances of these decidable constructs.

<pre> data Elem : (p : type -> Type) -> (x : type) -> (xs : List type) -> Type where Here : (prfSame : x = y) -> (prfSat : p y) -> Elem p x (y::xs) There : (rest : Elem p x xs) -> Elem p x (y::xs) </pre>	<pre> isElemSat : DecEq type => (f : (x : type) -> DecInfo (n x) (p x)) -> (x : type) -> (xs : List type) -> DecInfo (ElemNot n p x xs) (Elem p x xs) </pre>
<p>(a) Decidable Predicate.</p>	<p>(b) Decision Procedure.</p>

■ **Figure 15** Quantification over Lists.

The standard decidable predicate (*Dec*) for decision procedures is lossy when reporting errors, such that it is impossible to know at runtime why the procedure failed. Such predicates are analogous to the *Maybe* datatype. We need an *Either* equivalent to help report useful information when a decidable procedure fails: *Dec* needs to be decidedly informative. Constructive negation is an interesting area of research that supports positive information to be used when reporting errors [5]. Inspired by constructive negation we have used *DecInfo*, which is similar to the definition for decidable but is further indexed by a type that carries showable error messages as well as proofs of contradiction. We see this in the type signature for *isElemSat* in Figure 15, where *ElemNot* records why the procedure failed and requires an informative decision procedure: empty list, or element does not satisfy

8:20 Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

```

data Update : (o,n : type -> Type)
  -> (use : (x : type) -> (prf0 : o x) -> (y : type) -> (prfN : n y) -> Type)
  -> (old : List type) -> (prf : Elem o x old)
  -> (new : List type)
  -> Type

where
  UHere : {0 u : (x : type) -> (p0 : o x) -> (y : type) -> (pN : n y) -> Type}
    -> (use : u x pX y pY)
        -> Update o n u (x::xs) (Here Refl pX)
            (y::xs)
  UThere : (rest : Update o n u xs l ys)
    -> Update o n u (x::xs) (There l) (x::ys)

(a) Instructions.

update : (use : {x : type} -> (prf0 : o x) -> (y : type ** prfN : n y ** u x prf0 y prfN))
  -> (old : List type)
  -> (prf : Elem o x old)
  -> (new ** Update o n u old prf new)

(b) Procedure.

```

■ **Figure 16** Safe List Updating.

the provided predicate. Sadly `DecInfo` is not truly positive as it carries a *proof of void*. Making it *positively negative*, that is using only positive information to represent proofs and refutations, is ongoing work.

Using `Idris2` to construct our proof-of-concept type-checkers demonstrates the power of mechanising our proofs in a practical general purpose language that supports dependent types.

7 Evaluation

We investigated the efficacy of `CIRCUITS` and `CIRQTS` through creation of an illustrative testbench. The testbench was designed to illustrate how well-known circuits can contain implicit wiring decisions (checked against `CIRCUITS`), and that by using `CIRQTS` we can make the wiring explicit. Other examples chosen were inspired by various online Verilog tutorials and sought to test the ability of our type-systems to type-check existing designs (`CIRCUITS`) and to reason about quantitative wire usage for those designs (`CIRQTS`). The well-known designs chosen were flip flops, full and half adders, and gate-level multiplexers. We provided normal and linear variants of these well-known designs, and Table 1 presents salient modelling information comparing the two variants.

■ **Table 1** Salient Modelling Information for Core Netlists used during Benchmarking.

Netlist	Non-Linear				Linear			
	Inputs	Outputs	Wires	Gates	Inputs	Outputs	Wires	Gates
FlipFlopD	2	2	3	5	2	2	11	9
FullAdder	3	2	3	5	3	2	11	9
HalfAdder	2	2	0	2	2	2	4	4
Mux21	3	1	3	4	3	1	5	5

All examples presented were checked against: `CIRCUITS`, `CIRQTS`, and `Verilator`. We performed these checks for two reasons. First, we wanted to show that designs that were admitted by `CIRCUITS` but failed the soundness checker, then failed to type check using

CIRQTS, and that valid linear designs could also be checked using CIRCUITS. Second, Verilator is a well-known open source static analysis and simulation tool for SystemVerilog. Comparison against Verilator provides a validation step that CIRQTS is comparable to existing tooling and that, unlike commercial static analysis tools, can be incorporated into our supplied artefact due to its small installation footprint and permissive licence.

For all examples, we found that type-checking time was negligible, and comparable to Verilator, whether an error was found or not. From this experimentation we made the following observations.

We can Retrofit Linear Types onto Verilog NetLists

The gate-level syntax is the last intermediate representation before fabrication (or deployment to an FPGA). For this subset of Verilog chosen for our featherweight language, we have been successful in introducing linear wirings. From this subset the challenge will be how to promote linearity to the remainder of Verilog, and of course SystemVerilog.

Verbosely Made Implicit Wirings Explicit

The new primitives presented in Section 5 support valid weakening of linearity, specifically splitting and joining of wires. Such weakening, however, comes at the cost of design verbosity. Compare in Figure 17, for example, the implementations of a half-adder in both CIRCUITS and CIRQTS. For each wire split, or joined, new wires must be presented.

<pre> module Example(output logic sum, carry ,input logic a b); wire logic a1,a2,b1,b2; module Example(output logic sum, carry ,input logic a b); xor g1(sum,a,b); and g2(carry,a,b); endmodule; (a) CIRCUITS. </pre>	<pre> module Example(output logic sum, carry ,input logic a b); wire logic a1,a2,b1,b2; split sa(a1,a2,a); split sb(b1,b2,b); xor g1(sum,a1,b1); and g2(carry,a2,b2); endmodule; (b) CIRQTS. </pre>
--	---

■ **Figure 17** Half-Adder in both CIRCUITS and CIRQTS.

Although in the linear setting such redundant wires can be optimised away, their addition to the language makes it verbose. If we are to introduce such linear restrictions elsewhere in SystemVerilog, that is for synthesisable terms, the end user will be presented with a *needlessly* verbose language. Thus raising the question of how best to retrofit existing *HDLs* with quantitative types such that linearity does not get in the way. Perhaps we should not take this approach, and instead look at providing annotations as seen in other extensions to Verilog [48].

Some Ports are Linear but Some are More Linear Than Others

One aspect we found interesting is that CIRQTS performs more finegrained resource tracking for inputting n-dimensional bit-vectors when compared to Verilator. Although Verilator can identify unused inputs in designs, for logic and bit-vectors, it does not consistently report

8:22 Wiring Circuits Is Easy as $\{0, 1, \omega\}$, or Is It...

unused inputs for n-dimensional bit-vectors. Consider, for example, the following illustrative netlists presented in Figure 18. The netlist within Figure 18a *is* caught by Verilator as the output port (`out`) is not fully used. The netlist within Figure 18b *is not* caught by Verilator even though the input port (`bc`) is not fully used.

```
module Example(output logic[1:0][1:0] out, input logic a,b);  
  
    and n1(out[0][1],a,b);  
  
endmodule;
```

(a) Caught by Verilator.

```
module Example(output logic out, input logic[1:0][1:0] bc);  
  
    and n1(out, bc[0][1], bc[1][0]);  
  
endmodule;
```

(b) Not Caught by Verilator.

■ **Figure 18** Illustrative Netlists with sparsely used input and output ports.

We noted the potential discrepancy in Section 4.3 when discussing the stopping condition, that is when and how we check for linear usage. Such difference is because unused signals can be safely removed from the design. Specifically, inputting ports can be left unused if they are never connected to anything else in the entire design, and module outputs need to be driven as they could potentially be used later in the design to drive another part of the design.

Such differences in granularity of reporting between CIRQTS and Verilator does raise interesting questions about the semantics (veracity) of linear wiring. Whilst outputting ports must be linear in usage, inputting ports are affine. Linear usage restrictions are incredibly restrictive, terms must be used exactly once. In certain circuit designs it is okay to leave the input of a circuit dangling. Perhaps our quantitative type-system requires fine-tuning to ensure that we are affine for inputs, but linear for outputs, or even gradations to specify how much a signal must be used as seen in Granule? Although we could also extend our syntax to support “noops” primitives that *consumes* purposefully dangling wires. . . . That being said, such allowed dangling is very much a *behavioural aspect* that requires more information that we do not presently give in the type-system about the intended behaviour being realised. Our use of quantitative typing is purely structural and not behavioural. More descriptive behavioural typing is required, and that is a different kettle of fish entirely.

8 Related Work

Section 4 discussed the limitations of modelling linear wiring in existing *QTS*, and mentioned that Verilator performs an extrinsic graph-based check. This section looks to the wider world of *HDL* design and verification, and where our work resides.

8.1 Formal Models of Verilog

There have been several attempts at verifying the *behaviour* of Verilog designs [28, 31, 26]. Many of these works are limited by looking at the behaviour of modules in isolation and not their composition. CIRQTS complements these attempts by providing *a* formal account for wiring. Integrating our type-system into existing behavioural semantics would be advantageous.

8.2 BlueSpecVerilog

Verilog-like languages view hardware in the traditional sense of wiring up boxes/gates that communicate concurrently. BlueSpecVerilog (*BSV*) [32] takes a different route by allowing designers to take a slightly higher-level view of hardware and describe circuits as atomic actions on stateful elements i.e. registers. *BSV* builds upon SystemVerilog with strong types and known techniques lifted from the well-known functional language Haskell. Kami and Kôika are two rule-based hardware description languages that capture core behavioural semantics of *BSV* [11, 8]. Both languages are presented as *EDSLs* written in the Coq Theorem Prover. Our approach to wire-safety can compliment existing work relating to *BSV* by enforcing wiring decisions to be explicit rather than implicit. How our work intersects practically is an area for future investigation.

8.3 High-Level Synthesis

Similar to the realisation of Kami and Kôika, High-Level Synthesis (*HLS*) typically approaches hardware design as *EDSLs* in high-level languages such as Haskell and Scala that are then synthesised into hardware descriptions i.e. netlists. Popular *HDLs* that take this approach are PyRTL [13], Lava [20], Chisel [6], Delite [43], ReWrire [39], and Clash [42]. By embedding their work in an existing general-purpose language these languages can take full advantage of the host language’s eco-system to provide guarantees about program composition and behaviour. For instance *Wire Sorts* [12] is an extension to PyRTL to reason about wirings. Many of the approaches described here model circuit level designs as combinators, and treat hardware components as functions that can be translated directly to SystemVerilog. Our work intersects through provision of a means to account for how wires are generated and used in the synthesised designs.

More interestingly are *HDLs* embedded in languages that are *QTS* aware e.g. Haskell. It will be interesting to see how Linear Haskell can be used by the Lava and Clash to enforce linear wiring.

8.4 Cava

Coq + Lava (*Cava*) is an industrial research project² that looks at providing an *easy to verify HDL* for “network-style” low level circuits, rather than higher level processor designs as seen in *BSV*. *Cava* primarily looks to verify circuit behaviour. Much like Kami and Kôika, *Cava* is an *EDSL* in the Coq Theorem Prover. Interestingly, *Cava* and CIRCUITS/CIRQTS share similarities in that they both describe netlists. We differ in that *Cava* syntax is not verilog-like when representing wiring but ours is.

8.5 Formal Models for High-Level Synthesis

We end our discussion of related work by looking at theoretical models for hardware. The *Geometry of Synthesis* series [16, 19, 17, 15, 18] looks at leveraging *Geometry of Interaction* style interpretation as a way to better understand the behavioural and structural aspects of hardware design. Especially, they target a *HLS* combined with category theory to better tell the synthesis story. Fortunately, existing work (Section 8.1) has looked at the synthesis story for Verilog. We have not explored these stories to include *QTS*, we have short-circuited this story by jumping straight to the end. That being said we are, however, interested in exploring how our *QTS* can be included.

² <https://project-oak.github.io/silveroak/>

9 Conclusion

CIRQTS demonstrates that quantitative typing, specifically, the idea of linearity, is not just for general purpose programming languages. We can use *fancy types* to be reason about hardware design. With Verilog being so integral to hardware design, our efforts at formal verification (through typing) of the language itself has helped move existing extrinsic checks (as performed by static analyses and simulation, and also caught during synthesis) directly into the type-system, and become an integral part of the language itself. With the correct type-system, errors that were once caught late in the design process will now be caught earlier helping to increase design productivity and enhance the trustworthiness of developed designs.

A question remains though over what precise usage restrictions should be applied to hardware terms, and presented to the system designer. Is it the case that all terms should be restricted equally, or that some terms be restricted less equally than others? Moreover, we have not required designers to annotate their designs. Would it be better to extend/embed Verilog with linear annotations? If such restrictions are given, would some form of gradations *a la* Granule on terms be better for specifying usage? These are all exciting open areas of investigation.

A secondary area of interest is where we place usage information in CIRQTS. We have used usage *annotations* on binders, as is common in *QTS*. An alternative approach would be to index types directly with their usage, rather than as usage annotations on binders. We think this is an equally valid approach but will require further investigation over soundness and suitability.

Regardless, we have demonstrated how to *retrofit* a sub-structural type-system onto existing syntax modulo minor syntax elaboration. Retrofitting means that we do not need new languages that existing designs must be ported to. However, looking at what syntax changes are needed to make Verilog more conducive to linear wiring, or new primitives, is also a worthy area of future investigation. Our retrofitting approach also opens up future work on what else needs changing (in the type-system) when bringing quantitative typing back up the synthesis to higher-levels of abstraction. A key aspect of which will be design composition in the face of modules and interfaces, and the quasi-dependently-typed parameterisation of modules and interfaces.

Finally, within Idris2 the none-one-tonne rig [30] provides a distinction between code available at runtime ($1, \omega$) and its usage, and code only available at compiletime (0). We have used elements of *QTT* to reason about wire usage, but not their movement through the *fourth dimension*: time. Whereas we think of a program's journey as going first through the compiletime and then the runtime, SystemVerilog is a language where designs will journey through many more "times": design, testing, simulation, synthesis, placement & routing, and execution. A fascinating area of investigation would be to see if *QTT*, or *QTT*-like structures, can help to better reason about terms as they traverse through the many times of SystemVerilog. Can terms be stratified according to *when* in time they are expected to be? We hope so!

References

- 1 Guillaume Allais. Typing with Leftovers – A mechanization of Intuitionistic Multiplicative-Additive Linear Logic. In *23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary*, pages 1:1–1:22, 2017. doi:10.4230/LIPIcs.TYPES.2017.1.

- 2 Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. A type and scope safe universe of syntaxes with binding: their semantics and proofs. *PACMPL*, 2(ICFP):90:1–90:30, 2018. doi:10.1145/3236785.
- 3 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18–20, 2017*, pages 666–679. ACM, 2017. URL: <http://dl.acm.org/citation.cfm?id=3009866>, doi:10.1145/3009837.3009866.
- 4 Robert Atkey. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09–12, 2018*, pages 56–65, 2018. doi:10.1145/3209108.3209189.
- 5 Robert Atkey. Data Types with Negation. Extended Abstract (Talk Only) at Ninth Workshop on Mathematically Structured Functional Programming, Munich, Germany, 2nd April 2022, 2022. URL: <https://youtu.be/mZZjOKWCF4A>.
- 6 Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a Scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3–7, 2012*, pages 1216–1225. ACM, 2012. doi:10.1145/2228360.2228584.
- 7 Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158093.
- 8 Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of Bluespec: a core language for rule-based hardware design. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15–20, 2020*, pages 243–257. ACM, 2020. doi:10.1145/3385412.3385965.
- 9 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.
- 10 Edwin C. Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11–17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPICs*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ECOOP.2021.9.
- 11 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, 2017. doi:10.1145/3110268.
- 12 Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. Wire sorts: a language abstraction for safe hardware composition. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021*, pages 175–189. ACM, 2021. doi:10.1145/3453483.3454037.
- 13 Deeksha Dangwal, Georgios Tzimpragos, and Timothy Sherwood. Agile Hardware Development and Instrumentation With PyRTL. *IEEE Micro*, 40(4):76–84, 2020. doi:10.1109/MM.2020.2997704.
- 14 Peter Flake, Phil Moorby, Steve Golson, Arturo Salz, and Simon Davidmann. Verilog HDL and Its Ancestors and Descendants. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi:10.1145/3386337.
- 15 Dan R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17–19, 2007*, pages 363–375. ACM, 2007. doi:10.1145/1190216.1190269.

- 16 Dan R. Ghica. The Geometry of Synthesis – How to Make Hardware Out of Software. In *Mathematics of Program Construction – 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, pages 23–24, 2012. doi:10.1007/978-3-642-31113-0_3.
- 17 Dan R. Ghica and Alex I. Smith. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits. In Michael W. Mislove and Peter Selinger, editors, *Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2010, Ottawa, Ontario, Canada, May 6-10, 2010*, volume 265 of *Electronic Notes in Theoretical Computer Science*, pages 301–324. Elsevier, 2010. doi:10.1016/j.entcs.2010.08.018.
- 18 Dan R. Ghica and Alex I. Smith. Geometry of synthesis III: resource management through type inference. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 345–356. ACM, 2011. doi:10.1145/1926385.1926425.
- 19 Dan R. Ghica, Alex I. Smith, and Satnam Singh. Geometry of synthesis IV: compiling affine recursion into static hardware. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 221–233, 2011. doi:10.1145/2034773.2034805.
- 20 Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In Marco T. Morazán and Sven-Bodo Scholz, editors, *Implementation and Application of Functional Languages – 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers*, volume 6041 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2009. doi:10.1007/978-3-642-16478-1_2.
- 21 Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight Go. *Proc. ACM Program. Lang.*, 4(OOPSLA):149:1–149:29, 2020. doi:10.1145/3428217.
- 22 Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485494.
- 23 IEEE. *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*, IEEE Std 1800-2017 edition, February 2018. doi:10.1109/IEEESTD.2018.8299595.
- 24 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.
- 25 Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.
- 26 Wilayat Khan, Alwen Tiu, and David Sanán. VeriFormal: An Executable Formal Model of a Hardware Description Language. In Abhik Roychoudhury and Yang Liu, editors, *A Systems Approach to Cyber Security – Proceedings of the 2nd Singapore Cyber-Security R&D Conference (SG-CRC 2017), Singapore, February 21-22, 2017*, volume 15 of *Cryptology and Information Security Series*, pages 19–36. IOS Press, 2017. doi:10.3233/978-1-61499-744-3-19.
- 27 Wen Kokke, Jeremy G. Siek, and Philip Wadler. Programming language foundations in Agda. *Sci. Comput. Program.*, 194:102440, 2020. doi:10.1016/j.scico.2020.102440.
- 28 Andreas Lööw. Lutsig: a verified Verilog compiler for verified circuit development. In Cătălin Hrițcu and Andrei Popescu, editors, *CPP ’21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 46–60. ACM, 2021. doi:10.1145/3437992.3439916.
- 29 Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In Bruno C. d. S. Oliveira and Marcin Zalewski, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming, WGP 2010, Baltimore, MD, USA, September 27-29, 2010*, pages 1–12. ACM, 2010. doi:10.1145/1863495.1863497.

- 30 Conor McBride. I Got Plenty o' Nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World – Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. doi:10.1007/978-3-319-30936-1_12.
- 31 Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Rosu. A formal executable semantics of Verilog. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*, pages 179–188. IEEE Computer Society, 2010. doi:10.1109/MEMCOD.2010.5558634.
- 32 Rishiyur S. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*, pages 69–70. IEEE Computer Society, 2004. doi:10.1109/MEMCOD.2004.1459818.
- 33 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *PACMPL*, 3(ICFP):110:1–110:30, 2019. doi:10.1145/3341714.
- 34 Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In Wei-Ngan Chin and Aquinas Hobor, editors, *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, page 1. ACM, 2012. doi:10.1145/2318202.2318203.
- 35 Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for imperative languages. *Proc. ACM Program. Lang.*, 2(POPL):16:1–16:34, 2018. doi:10.1145/3158104.
- 36 Dimitri Racordon and Didier Buchs. Featherweight Swift: a Core calculus for Swift's type system. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 140–154. ACM, 2020. doi:10.1145/3426425.3426939.
- 37 John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. doi:10.1023/A:1010027404223.
- 38 John C. Reynolds. Definitional Interpreters Revisited. *High. Order Symb. Comput.*, 11(4):355–361, 1998. doi:10.1023/A:1010075320153.
- 39 Thomas N. Reynolds, Adam M. Procter, William L. Harrison, and Gerard Allwein. A core calculus for secure hardware: its formal semantics and proof system. In Jean-Pierre Talpin, Patricia Derler, and Klaus Schneider, editors, *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, Vienna, Austria, September 29 – October 02, 2017*, pages 122–131. ACM, 2017. doi:10.1145/3127041.3127048.
- 40 Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 284–298. ACM, 2020. doi:10.1145/3372885.3373818.
- 41 Jeremy Siek. Type Safety in Three Easy Lemmas. Online, May 2013. URL: <https://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html>.
- 42 Gerard J. M. Smit, Jan Kuper, and Christiaan P. R. Baaij. A mathematical approach towards hardware design. In Peter M. Athanas, Jürgen Becker, Jürgen Teich, and Ingrid Verbauwhede, editors, *Dynamically Reconfigurable Architectures, 11.07. – 16.07.2010*, volume 10281 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, 2010. doi:10.4230/DagSemProc.10281.3.
- 43 Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embedded Comput. Syst.*, 13(4s):134:1–134:25, 2014. doi:10.1145/2584665.

- 44 Philip Wadler. Linear Types can Change the World! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.
- 45 David Walker. *Advanced Topic in Types and Programming Languages*, chapter Substructural Type Systems, pages 3–43. The MIT Press, 2004.
- 46 James Wood and Robert Atkey. A Linear Algebra Approach to Linear Metatheory. In Ugo Dal Lago and Valeria de Paiva, editors, *Proceedings Second Joint International Workshop on Linearity & Trends in Linear Logic and Applications, Linearity & TLLA @ IJCAR-FSCD 2020, Online, 29-30 June 2020*, volume 353 of *EPTCS*, pages 195–212, 2020. doi:10.4204/EPTCS.353.10.
- 47 Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.
- 48 Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2694344.2694372.

VeriFx: Correct Replicated Data Types for the Masses

Kevin De Porre ✉ 🏠 

Vrije Universiteit Brussel, Belgium

Carla Ferreira ✉ 🏠 

NOVA School of Science and Technology, Caparica, Portugal

Elisa Gonzalez Boix ✉ 🏠 

Vrije Universiteit Brussel, Belgium

Abstract

Distributed systems adopt weak consistency to ensure high availability and low latency, but state convergence is hard to guarantee due to conflicts. Experts carefully design replicated data types (RDTs) that resemble sequential data types and embed conflict resolution mechanisms that ensure convergence. Designing RDTs is challenging as their correctness depends on subtleties such as the ordering of concurrent operations. Currently, researchers manually verify RDTs, either by paper proofs or using proof assistants. Unfortunately, paper proofs are subject to reasoning flaws and mechanized proofs verify a formalization instead of a real-world implementation. Furthermore, writing mechanized proofs is reserved for verification experts and is extremely time-consuming. To simplify the design, implementation, and verification of RDTs, we propose VeriFx, a specialized programming language for RDTs with *automated* proof capabilities. VeriFx lets programmers implement RDTs atop functional collections and express correctness properties that are verified automatically. Verified RDTs can be transpiled to mainstream languages (currently Scala and JavaScript). VeriFx provides libraries for implementing and verifying Conflict-free Replicated Data Types (CRDTs) and Operational Transformation (OT) functions. These libraries implement the general execution model of those approaches and define their correctness properties. We use the libraries to implement and verify an extensive portfolio of 51 CRDTs, 16 of which are used in industrial databases, and reproduce a study on the correctness of OT functions.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Computing methodologies → Distributed programming languages; Theory of computation → Distributed algorithms

Keywords and phrases distributed systems, eventual consistency, replicated data types, verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.9

Related Version *Previous Version:* <https://arxiv.org/abs/2207.02502>

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.9.2.19>

Funding *Kevin De Porre:* Funded by the Research Foundation - Flanders. Project number 1S98519N. *Carla Ferreira:* Partly funded by EU Horizon Europe under Grant Agreement no. 101093006 (TaRDIS), and FCT-Portugal under grants UIDB/04516/2020 and PTDC/CCI-INF/32081/2017.

Acknowledgements The authors would like to thank Nuno Preguiça, Carlos Baquero, and Imine Abdessamad for their early feedback on this work.

1 Introduction

Replication is essential to modern distributed systems as it enables fast access times and improves the system's overall scalability, availability, and fault tolerance. When data is replicated across machines, replicas must be kept consistent to some extent. When facing network partitions, replicas cannot remain consistent while also accepting reads and writes, a consequence of the CAP theorem [17, 18, 39]. Programmers thus face a trade-off between



© Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 9; pp. 9:1–9:45



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

consistency and availability. Keeping replicas strongly consistent induces high latencies, poor scalability, and reduced availability since updates must be coordinated, e.g. using a consensus algorithm. By relaxing the consistency guarantees, latencies can be reduced and the overall availability improved, but users may observe temporary inconsistencies between replicas.

Distributed systems increasingly adopt weak consistency models. However, concurrent operations may lead to conflicts which must be solved in order to guarantee state convergence. Consider the case of collaborative text editors. When a user edits a document, the operation is immediately applied on the local replica and propagated asynchronously to other replicas. Since concurrent edits are applied in different orders at different replicas, states can diverge.

To ensure convergence, Ellis and Gibbs [25] proposed a technique called Operational Transformation (OT) that modifies incoming operations against previously executed concurrent operations such that the modified operation preserves the intended effect. Much work focused on designing OT functions for collaborative text editing [25, 34, 64, 69, 72], but most tombstone-free transformation functions (some with mechanized proofs) are wrong [34, 49, 62].

Since conflict resolution is hard [2, 41, 68], researchers now focus on designing replicated data types (RDTs) that serve as building blocks for the development of highly available distributed systems. Such RDTs resemble sequential data types (e.g. counters, sets) but include conflict resolution strategies that guarantee convergence.

Conflict-free Replicated Data Types (CRDTs) [68] are a widely adopted family of RDTs that leverage mathematical properties (such as commutative operations) to avoid conflicts by design. However, designing new RDTs is difficult [42] and even seasoned researchers miss subtle corner cases for basic data structures such as maps [40]. Currently, researchers and practitioners propose new or improved RDT designs [2, 8, 14, 19, 38, 41, 66–68] and include a formal specification or pseudo code of the RDT with a manual proof of convergence, mostly paper proofs. Unfortunately, paper proofs are subject to reasoning flaws.

To avoid the pitfalls of paper proofs, Zeller et al. [79], Gomes et al. [27], and Nieto et al. [59] propose formal frameworks to verify the correctness of CRDTs using proof assistants. However, these frameworks use abstract specifications that are disconnected from actual implementations (e.g. Akka’s CRDT implementations in Scala). Hence, a particular implementation may be flawed, even if the specification was proven to be correct.

While interactive proofs are more convincing (as the proof logic is machine-checked), they require significant programmer intervention which is time-consuming and reserved to experts [45, 60]. Recent works try to automate (part of) the verification process of CRDTs. Nagar and Jagannathan [56] automatically verify CRDTs under different consistency models but require a first-order logic specification of the CRDT’s operations that is cumbersome and error-prone. Liu et al. [54] extend Liquid Haskell [75] to verify CRDTs but significant parts need to be proven manually due to the way how Liquid Haskell encodes user-defined functions in SMT. For example, their Map CRDT required more than 1000 lines of proof code. We conclude that developing RDTs is reserved for experts in distribution and verification.

To simplify the development of RDTs, we propose VeriFx, a specialized functional object-oriented programming language for designing, implementing, and *automatically* verifying RDTs. The main challenge behind VeriFx’s design consists in striking a good balance between expressiveness and automated verification. We designed VeriFx to support familiar, high-level language constructs that are suited to implement RDTs, without breaking automated verification. To *implement* RDTs, VeriFx provides extensive functional collections including tuples, sets, maps, vectors, and lists. These collections are immutable which is said to be desirable for the implementation of RDTs and their integration in distributed systems [30]. To *verify* RDT implementations, VeriFx features a novel proof construct that enables

programmers to express correctness properties. For each proof, VeriFx automatically derives proof obligations and discharges them using SMT solvers. This is possible because VeriFx efficiently encodes all functional collections and their operations using the Combinatory Array Logic [23] for SMT solvers, which is decidable. This enables VeriFx to automatically verify complex RDTs built atop these collections. VeriFx provides libraries that ease the implementation and verification of CRDT and OT data types. Internally, these libraries use the proof construct to define the necessary correctness properties. Verified RDTs can be transpiled to one of the supported target languages (currently Scala and JavaScript).

VeriFx is reminiscent of existing object-oriented languages (like Scala) and demonstrates that it is possible to automatically verify real-world RDT implementations without requiring separate specifications. This avoids mismatches between the implementation and the specification and simplifies software maintenance. We argue that the ability to implement RDTs and automatically verify them in the *same* language allows programmers to catch mistakes early during the development process.

To demonstrate the applicability of VeriFx, we implemented and *automatically* verified 51 CRDTs, including well-known CRDTs [2, 8, 14, 40, 66, 67] and new variants. 50 of these CRDTs were verified in a matter of seconds and one could not be verified due to its recursive nature. The CRDTs we verified feature highly optimized designs and many are used in industrial databases such as Riak [12], Cassandra [73], and AntidoteDB [1]. We also applied VeriFx to OT and verified *all* transformation functions described by Imine et al. [34] and some unpublished designs [33].

In summary, we make the following contributions:

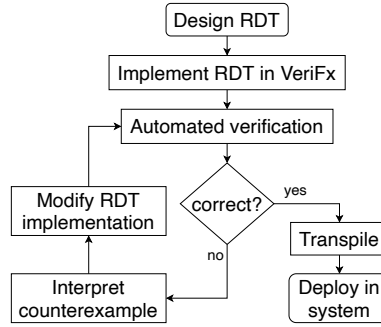
- VeriFx, the first high-level programming language that enables programmers to implement RDTs by composing functional collections, express correctness properties about those RDTs within the same language, and automatically verify those properties. The novelty consists of carefully crafting the language such that every language construct is efficiently encoded without breaking automated verification.
- We devise VeriFx libraries that simplify the implementation of CRDT and OT data types and automatically verify the necessary correctness properties.
- We give the first fully automated and mechanized proofs for 51 CRDTs, including the state-based and operation-based CRDTs proposed by Shapiro et al. [67], delta state-based CRDTs proposed by Almeida et al. [2], pure op-based CRDTs proposed by Baquero et al. [8], and many more. To the best of our knowledge, this is the most extensive treatment of verified RDTs to date. Prior efforts [27, 54, 59, 79] verified only a few CRDTs due to the labour-intensive nature of the verification process.
- We reproduce the study of Imine et al. [34] regarding the verification of OT functions.

2 Motivation

To motivate the need for a language with automated proof capabilities, consider a distributed system in Scala with replicated data on top of Akka's highly-available distributed key-value store [52]. The store provides built-in CRDTs, e.g. sets, counters, etc. However, our system requires a Two-Phase Set (2PSet) CRDT [67] that is not provided by Akka. We thus need to implement it and verify our implementation.

For the implementation, we can take the specification from Shapiro et al. [67]. For the verification, we typically need a complete formalization of the implementation and its correctness properties which can then be proven manually using proof assistants. The resulting interactive proofs are complex and require considerable expertise. For example, Nieto et al. [59]'s implementation of a 2PSet in OCaml is only 25 LoC but its specification in Coq is 80 LoC and requires an additional 73 LoC to verify.

Alternatively, programmers could resort to Liu et al. [54]’s extension of Liquid Haskell that automates part of the verification process. However, non-trivial RDTs still require significant manual proof efforts: 200+ LoC for a replicated set and 1000+ LoC for a map [54]. Thus, we cannot reasonably assume that programmers have the time nor the skills to manually verify their implementation [45,60].



■ **Figure 1** Envisioned workflow.

We argue that verification needs to be fully automatic to be accessible to non-experts. Figure 1 shows the workflow for developing RDTs using VeriFx, our novel language with a syntax reminiscent of Scala. Programmers start from a new or existing RDT design and implement it in VeriFx which verifies the implementation automatically without requiring a separate formalization. If the implementation is not correct, VeriFx returns a counterexample in which the replicas diverge. After interpreting the counterexample, the programmer needs to fix the RDT implementation and verify it again. This iterative process repeats until the implementation is correct. Verified RDT implementations can be transpiled to a mainstream language (e.g. Scala) and deployed in an actual system.

Our envisioned workflow verifies RDT implementations before deployment. Moreover, our workflow benefits from a feedback loop allowing programmers to correct implementations based on concrete counterexamples. In contrast, traditional verification techniques such as interactive theorem provers do not provide such feedback; when programmers fail to verify a property, they do not know if the implementation is flawed or if the chosen proof strategy is not suited. Similarly, Liquid Haskell [75] may fail to verify a property and raise a type error without providing additional information as to why the refinement type is not met. Next, we illustrate each step of our workflow by means of an existing 2PSet design in which VeriFx uncovered a bug. The corrected version was then transpiled to Scala, and deployed on Akka.

2.1 Design and Implementation

Specification 1 shows the design of the 2PSet CRDT taken from Shapiro et al. [67] unaltered. The 2PSet is a state-based CRDT whose state (the A and R sets) thus forms a join semilattice, i.e. a partial order \leq_v with a least upper bound (LUB) \sqcup_v for all states. Elements are added to the 2PSet by adding them to the A set and removed by *adding* them to the R set. An element is in the 2PSet if it is in A and not in R . Hence, removed elements can never be added again. Replicas are merged by computing the LUB of their states, which in this case is the union of their respective A and R sets.

The `compare(S, T)` operation checks if $S \leq_v T$ and is used to define state equivalence [68]: $S \equiv T \iff S \leq_v T \wedge T \leq_v S$. Since state equivalence is defined in terms of \leq_v on the lattice, replicas may be considered equivalent even though they are not identical. This is relevant

■ Specification 1 2PSet CRDT

taken from Shapiro et al. [67].

```

1: payload set  $A$ , set  $R$ 
2:   initial  $\emptyset, \emptyset$ 
3: query lookup (element  $e$ ) : boolean  $b$ 
4:   let  $b = (e \in A \wedge e \notin R)$ 
5: update add (element  $e$ )
6:    $A := A \cup \{e\}$ 
7: update remove (element  $e$ )
8:   pre lookup( $e$ )
9:    $R := R \cup \{e\}$ 
10: compare ( $S, T$ ) : boolean  $b$ 
11:   let  $b = (S.A \subseteq T.A \vee S.R \subseteq T.R)$ 
12: merge ( $S, T$ ) : payload  $U$ 
13:   let  $U.A = S.A \cup T.A$ 
14:   let  $U.R = S.R \cup T.R$ 

```

■ Listing 1 2PSet implementation in VeriF_x.

```

1 class TwoPSet[V](added: Set[V], removed: Set[V])
2   extends CvRDT[TwoPSet[V]] {
3   def lookup(element: V) =
4     this.added.contains(element) &&
5     !this.removed.contains(element)
6   def add(element: V) =
7     new TwoPSet(this.added.add(element), this.removed)
8   def remove(element: V) =
9     new TwoPSet(this.added, this.removed.add(element))
10  def compare(that: TwoPSet[V]) =
11    this.added.subsetOf(that.added) ||
12    this.removed.subsetOf(that.removed)
13  def merge(that: TwoPSet[V]) =
14    new TwoPSet(this.added.union(that.added),
15               this.removed.union(that.removed))
16  }

```

for CRDTs that keep additional information. For example, CRDTs often use a Lamport clock [43] together with unique replica identifiers to generate globally unique IDs. Every replica identifier is different and is not part of the lattice even though it is part of the state.

Listing 1 shows the implementation of the 2PSet CRDT in VeriF_x, which is a straightforward translation of Specification 1. The `TwoPSet` class is polymorphic in the type of values it stores. It defines the `added` and `removed` fields which correspond to the A and R sets respectively. The `add` and `remove` methods return an updated copy of the state. The class extends the `CvRDT` trait¹ provided by VeriF_x's CRDT library (explained in Section 5.1). This trait requires the class to implement the `compare` and `merge` methods.

2.2 Verification

We now verify our 2PSet implementation in VeriF_x. State-based CRDTs guarantee convergence if the merge function is idempotent, commutative, and associative [68]. VeriF_x provides several `CvRDTProof` traits which encode these correctness conditions (explained later in Section 5.1). To verify the `TwoPSet`, we define a `TwoPSetProof` object that extends the `CvRDTProof1` trait (where 1 is the rank). The trait takes as argument the type constructor of the CRDT we want to verify (i.e. `TwoPSet`):

```
object TwoPSetProof extends CvRDTProof1[TwoPSet]
```

The `TwoPSetProof` object inherits automated correctness proofs for the polymorphic `TwoPSet` CRDT. When executing this object, VeriF_x will automatically try to verify those proofs. In this case, VeriF_x proves that the `TwoPSet` guarantees convergence (independent of the type of elements it holds), according to the notion of state equivalence that is derived from `compare`. However, VeriF_x warns the user that the proof for state equivalence fails, which means that the derived notion of equivalence does not correspond to structural equality. As explained before, this may be normal in some CRDT designs but it requires further investigation.

VeriF_x provides the following counterexample for the equivalence proof:

```
enum V { v }
val s: TwoPSet[V] = TwoPSet({v}, {})
val t: TwoPSet[V] = TwoPSet({v}, {v})
```

¹ VeriF_x traits can declare abstract methods and fields, and provide default implementations for methods.

■ **Listing 2** Transpiled 2PSet in Scala.

```

1 case class TwoPSet[V](added: Set[V], removed: Set[V])
  extends CvRDT[TwoPSet[V]] { // CvRDT trait provided
  by our CRDT library is also compiled to Scala
2 def lookup(element: V) = this.added.contains(element) &&
  !this.removed.contains(element)
3
4 def add(element: V): TwoPSet[V] =
  TwoPSet[V](this.added + element, this.removed)
5
6 def remove(element: V): TwoPSet[V] =
  TwoPSet[V](this.added, this.removed + element)
7
8 def compare(that: TwoPSet[V]): Boolean =
  this.added.subsetOf(that.added) &&
  this.removed.subsetOf(that.removed)
9
10
11 def merge(that: TwoPSet[V]): TwoPSet[V] =
12   TwoPSet[V](this.added.union(that.added),
13             this.removed.union(that.removed)) }

```

■ **Listing 3** Modified 2PSet implementation for integration with Akka’s distributed key-value store.

```

1 @SerialVersionUID(1L)
2 case class TwoPSet[V](
3   added: Set[V], removed: Set[V])
  extends CvRDT[TwoPSet[V]] with
  ReplicatedData with Serializable {
4   type T = TwoPSet[V]
5   // The remainder of the implementation
  is unchanged
6 }

```

The counterexample defines an enumeration V containing a single value v . It then defines two instances s and t of a `TwoPSet[V]` that are considered equivalent $s \equiv t$ (according to the definition of `compare`) but are not structurally equivalent $s \neq t$. These two instances should indeed not be considered equivalent since $v \in s$ but $v \notin t$ according to `lookup`. Looking back at Spec. 1, we notice that the original specification of `compare` from Shapiro et al. [67] defines replica s to be smaller or equal to replica t iff $s.A \subseteq t.A$ or $s.R \subseteq t.R$. Since $s.A = t.A$ it follows that $s \leq_v t \wedge t \leq_v s$ and thus they are considered equal ($s \equiv t$) without even considering the removed elements (i.e. the R sets). Based on this counterexample, we modify `compare` to consider both the A sets and the R sets:

```

def compare(that: TwoPSet[V]) =
  this.added.subsetOf(that.added) && this.removed.subsetOf(that.removed)

```

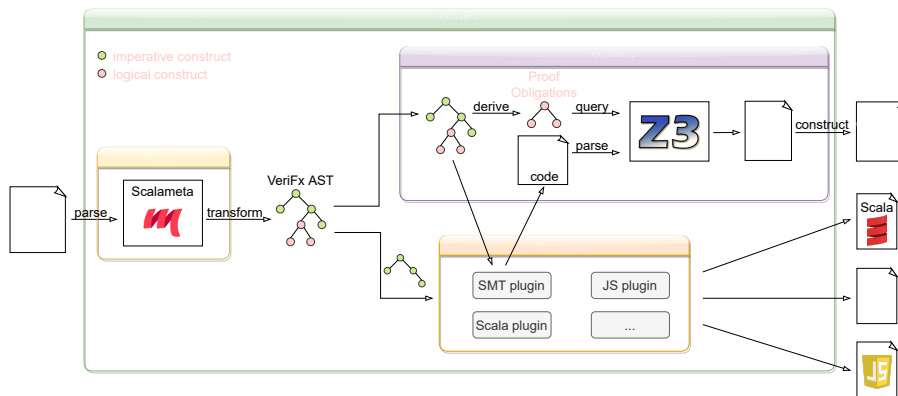
We verify the implementation again and VeriFx proves that this modified implementation still guarantees convergence and that the definition of equality that is derived from `compare` now corresponds to structural equality, i.e. $s \equiv t \iff s = t$.

This example showcases the importance of automated verification as it detected an error in the specification that would have percolated to the implementation. We completed the verification of the 2PSet CRDT in VeriFx without providing any verification-specific code.

2.3 Deployment

The final step in our workflow consists of automatically transpiling the CRDT implementation from VeriFx to Scala and integrating the CRDT in our distributed application which uses Akka’s distributed key-value store. Listing 2 shows the transpiled implementation of the 2PSet in Scala. To store the RDT in Akka’s distributed key-value store, we need to perform two manual modifications which are shown in Listing 3. First, the RDT must extend Akka’s `ReplicatedData` trait (Line 3) which requires at least the definition of a type member T corresponding to the actual type of the CRDT (Line 4) and a `merge` method for CRDTs of that type (which we already have). Second, the RDT must be serializable. For simplicity, we use Java’s built-in serializer². Hence, it suffices to extend the `Serializable` trait (Line 3) and to annotate the class with a serial version (Line 1). After applying these modifications, our verified `TwoPSet` can be stored in Akka’s distributed key-value store and will automatically be replicated across the cluster and be kept eventually consistent.

² In production it is safer and more efficient to implement a custom serializer [53], e.g. with Protobuf [28].



■ **Figure 2** VeriFx’s plugin architecture.

3 The VeriFx Language

The goal of this work is to build a familiar high-level programming language that is suited to implement RDTs and *automatically* verify them. The main challenge is to efficiently encode every feature of the language without breaking automatic verification. The result of this exercise is VeriFx, a functional object-oriented programming language with Scala-like syntax and a type system that resembles Featherweight Generic Java [32]. VeriFx features a novel proof construct to express correctness properties about programs. For every proof construct a proof obligation is derived that is discharged automatically by an SMT solver (cf. Section 4).

VeriFx advocates for the object-oriented programming paradigm as it is widespread across programmers and fits the conceptual representation of replicated data as “shared” objects. The functional aspect of VeriFx, in particular its immutable collections, makes it suitable for implementing and integrating RDTs in distributed systems, as argued by Helland [30].

The remainder of this section is organized in three parts. First, we give an overview of VeriFx’s architecture. Second, we define its syntax. Third, we describe its functional collections. VeriFx’s type system is described in Appendix A.

3.1 Overall Architecture

Figure 2 provides an overview of VeriFx’s architecture. VeriFx programs consist of imperative code and proof code (i.e. logic statements). VeriFx uses Scala Meta [65] to parse VeriFx source code into an AST representing the program. This is possible because every piece of VeriFx code is valid Scala syntax (but not necessarily semantically correct).

The AST representing a VeriFx program can be verified or transpiled to other languages. Transpilation is done by the compiler which features *compiler plugins*. These plugins dictate the compilation of the AST to the target language. Currently, VeriFx comes with compiler plugins for Scala, JavaScript, and SMT-LIB [74], a standardized language for SMT solvers. Support for other languages can be added by implementing a compiler plugin for them.

To verify the proofs that are defined by a VeriFx program, the verifier derives the necessary proof obligations from the AST. VeriFx then compiles the program to SMT-LIB and automatically discharges the proof obligations using the Z3 SMT solver [22]. For every proof, the outcome is: *accepted*, *rejected*, or *unknown*. Accepted means that the property holds, rejected means that a counterexample was found for which the property does not hold, and unknown means that the property could not be verified within a certain time

L	$::= \text{class } C \langle \overline{X} \rangle (\overline{v} : \overline{T}) \{ \overline{M} \}$	M	$::= \text{def } m \langle \overline{X} \rangle (\overline{x} : \overline{T}) : T = e$
	$\text{class } C \langle \overline{X} \rangle (\overline{v} : \overline{T}) \text{ extends } I \langle \overline{P} \rangle \{ \overline{M} \}$	T	$::= \text{int} \mid \text{string} \mid \text{bool} \mid C \langle \overline{T} \rangle$
J	$::= \text{object } O \{ \overline{A} \}$		$I \langle \overline{T} \rangle \mid E \langle \overline{T} \rangle \mid \overline{T} \rightarrow T$
	$\text{object } O \text{ extends } I \langle \overline{T} \rangle \{ \overline{A} \}$	e	$::= \text{num} \mid \text{str} \mid \text{true} \mid \text{false} \mid x \mid !e$
F	$::= \text{trait } I \langle \overline{X} <: \overline{T} \rangle \{ \overline{B} \}$		$e \oplus e \mid e \otimes e \mid e.v \mid e.m \langle \overline{T} \rangle (\overline{e})$
	$\text{trait } I \langle \overline{X} <: \overline{T} \rangle \text{ extends } I \langle \overline{P} \rangle \{ \overline{B} \}$		$\text{val } x : T = e \text{ in } e \mid (\overline{x} : \overline{T}) \Rightarrow e \mid e(\overline{e})$
N	$::= \text{enum } E \langle \overline{X} \rangle \{ K \langle \overline{v} : \overline{T} \rangle \}$		$\text{if } e \text{ then } e \text{ else } e$
A	$::= M \mid R$		$\text{new } C \langle \overline{T} \rangle (\overline{e}) \mid \text{new } K \langle \overline{T} \rangle (\overline{e})$
B	$::= \text{valD} \mid \text{methodD} \mid M \mid R$		$e \text{ match } \{ \text{case } \overline{r} \Rightarrow \overline{e} \}$
R	$::= \text{proof } p \langle \overline{X} \rangle \{ e \}$		$\text{forall } (\overline{x} : \overline{T}). e \mid \text{exists } (\overline{x} : \overline{T}). e$
valD	$::= \text{val } x : T$		$e \Longrightarrow e$
methodD	$::= \text{def } m \langle \overline{X} \rangle (\overline{x} : \overline{T}) : T$	r	$::= K \langle \overline{x} \rangle \mid x \mid _$

■ **Figure 3** VeriF_x syntax. The metavariable C ranges over class names; O ranges over object names; I ranges over trait names; E ranges over enumeration names; K ranges over constructor names of enumerations; T , P and Q range over types; X and Y range over type variables; v ranges over field names; x and y range over parameter and variable names; m ranges over method names; p ranges over proof names; and e ranges over expressions. An overline, e.g. \overline{X} , denotes zero or more. A dashed overline, e.g. $\overline{\overline{X}}$, denotes one or more.

frame (which is configurable). When a proof is rejected by Z3, VeriF_x constructs a high-level counterexample that consists of concrete assignments of values to variables that violate the given property. Note that VeriF_x can automatically verify application-specific properties because it derives the proof obligations from the program itself.

3.2 Syntax

Figure 3 defines the syntax of VeriF_x. VeriF_x programs consist of one or more statements which can be the definition of an object O , a class $C \langle \overline{X} \rangle$, a trait $I \langle \overline{X} \rangle$, or an enumeration $E \langle \overline{X} \rangle$. Objects, classes, enumerations, and traits can be polymorphic and inherit from a single trait (except enumerations). Objects define zero or more methods and proofs. Classes contain zero or more fields and (polymorphic) methods. Traits can declare values and methods that need to be provided by concrete classes extending the trait, and define (polymorphic) methods and proofs. Traits can express upper bounds on their type parameters to restrict possible extensions. Enumerations define one or more constructors, each containing zero or more fields. Programmers can deconstruct enumerations by pattern matching on them.

Unique to VeriF_x is its proof construct which is defined by a name and a (well-typed) boolean expression that expresses the property that must be verified. A proof is accepted if its body always evaluates to true, otherwise it is rejected; when rejected, VeriF_x provides a concrete counterexample for which the property does not hold. Proofs can be polymorphic, allowing properties to be proved for all possible type instantiations. Polymorphic proofs are useful to prove that a polymorphic RDT converges independent of its type of values.

VeriF_x supports a variety of expressions, including literal values, arithmetic \oplus and boolean operations \otimes , negation, field accesses, and method calls, variable definitions, if tests, anonymous functions and function calls, class and enum instantiations, pattern matching, quantified formulas, and logical implication. Functions are *first-class* and take at least one argument. Nullary functions can be expressed as constants.

Single inheritance is supported from traits to foster code re-use but some restrictions are imposed. E.g, the arguments of a class method need to be concrete (cannot be of a trait type) because proofs about these methods require reasoning about all subtypes but these may not necessarily be known at compile time. In contrast, enumerations are supported because their constructors are fixed and known at compile time.

Tuple<A, B>		Map<K, V>	
+ fst : A + snd : B + Tuple(fst: A, snd: B) : Tuple<A, B>		+ Map() : Map<K, V> + add(k: K, v: V) : Map<K, V> + remove(k: K) : Map<K, V> + contains(k: K) : bool + get(k: K) : V + getOrElse(k: K, default: V) : V + keys() : Set<K> + values() : Set<V> + bijective() : bool + map<W>(f: (K, V) => W) : Map<K, W> + mapValues<W>(f: V => W) : Map<K, W> + filter(p: (K, V) => bool) : Map<K, V> + zip<W>(m: Map<K, W>) : Map<K, Tuple<V, W>> + combine(m: Map<K, V>, f: (V, V) => V) : Map<K, V> + forall(p: (K, V) => bool) : bool + exists(p: (K, V) => bool) : bool + toSet() : Set<Tuple<K, V>>	
Set<V>		Vector<V>	
+ Set() : Set<V> + add(e: V) : Set<V> + remove(e: V) : Set<V> + contains(e: V) : bool + isEmpty() : bool + nonEmpty() : bool + union(s: Set<V>) : Set<V> + diff(s: Set<V>) : Set<V> + intersect(s: Set<V>) : Set<V> + subsetOf(that: Set[V]) : bool + map<W>(f: V => W) : Set<W> + filter(p: V => bool) : Set<V> + forall(p: V => bool) : bool + exists(p: V => bool) : bool		+ Vector() : Vector<V> + get(idx: Int) : V + write(idx: Int, value: V) : Vector<V> + append(value: V) : Vector<V> + map<W>(f: V => W) : Vector<W> + zip<W>(v: Vector<W>) : Vector<Tuple<V, W>> + forall(p: V => bool) : bool + exists(p: V => bool) : bool	
		List<V>	
		+ List() : List<V> + get(idx: Int) : V + insert(idx: Int, value: V) : List<V> + delete(idx: Int) : List<V> + map<W>(f: V => W) : List<W> + zip<W>(l: List<W>) : List<Tuple<V, W>> + forall(p: V => bool) : bool + exists(p: V => bool) : bool	

■ **Figure 4** An overview of VeriFx’s built-in functional collections.

3.3 Functional Collections

VeriFx features built-in collections for tuples, sets, maps, vectors, and lists. Every operation of these collections are verifiable and can be arbitrarily composed to build custom RDTs. All collections are immutable, “mutators” thus return an updated copy of the object. Figure 4 provides an overview of the interface exposed by these collections, which is heavily inspired by functional programming.

Sets. Support the typical set operations and can be mapped over or filtered using user-provided functions. The `forall` and `exists` methods check if a given predicate holds for all (respectively for at least one) element of the set.

Maps. Associate keys to values. Support adding key-value pairs, removing keys, and fetching the value associated with a key. The `keys` (resp. `values`) method returns a set containing all keys (resp. values) in the map. The `bijective` method checks if there is a one-to-one correspondence between keys and values. Maps support well-known functional operations; `zip` returns a map of tuples containing only the keys that are present in both maps and stores their values in a tuple; `combine` returns a map containing *all* entries from both maps, using a user-provided function `f` to combine values that are present in both maps.

Vectors. Represent a sequence of elements that are indexed from 0 to `size-1`. Elements can be written to a certain index which will overwrite the existing value at that index. One can append a value to the vector which will write that value at index `size`, thereby, making the vector grow. Like sets and maps, programmers can map functions over vectors, zip vectors, and check predicates for all or for one element of a vector.

Lists. Represent a sequence of elements in a linked list. Unlike vectors, `insert` does not overwrite the existing value at that index. Instead, the existing value at that index and all subsequent values are moved one position to the right. Elements can also be deleted from a list, making the list shrink.

$$\begin{array}{l}
T ::= \text{int} \mid \text{string} \mid \text{bool} \qquad G ::= \text{adt } A(\overline{X})\{K(\overline{v} : \overline{T})\} \qquad C ::= \text{const } x \ T \quad R ::= \text{assert } e \\
\mid \text{Array}(\overline{T}, T) \mid A(\overline{T}) \mid S(\overline{T}) \quad e ::= e[\overline{e}] \mid e[\overline{e}] := e \mid \lambda(\overline{x} : \overline{T}).e \quad D ::= \text{sort } S \ i \quad H ::= \text{check}() \\
F ::= \text{fun } f(\overline{X})(\overline{x} : \overline{T}) : T = e \qquad \mid \forall(\overline{x} : \overline{T}).e \mid \exists(\overline{x} : \overline{T}).e \mid \dots
\end{array}$$

■ **Figure 5** Core SMT syntax. The metavariable S ranges over user-declared sorts; A ranges over names of algebraic data types (ADTs); K ranges over ADT constructor names; X ranges over type variables; v ranges over field names; f ranges over function names; T ranges over types; x ranges over variable names; e ranges over expressions; and i ranges over integers.

4 Automated Verification

VeriF_x leverages SMT solvers to enable automated verification. Such solvers try to (automatically) determine whether or not a given formula is satisfiable. Modern SMT solvers support various specialized theories (for bit vectors, arrays, etc.) and are very powerful if care is taken to encode programs efficiently using these theories. However, SMT-LIB [74], the language of SMT solvers, is low-level and is not meant to be used directly by programmers to verify high-level programs. Instead, semi-automatic program verification usually involves implementing the program in an Intermediate Verification Language (IVL) which internally compiles to SMT-LIB to discharge the proof obligations using an appropriate SMT solver. IVLs like Dafny [44], Spec# [11], and Why3 [26] are designed to be general-purpose but this breaks automated verification since programmers need to specify preconditions and postconditions on methods, loop invariants, etc.

VeriF_x can be seen as a specialized high-level IVL that was carefully designed such that every feature has an efficient SMT encoding; leaving out features that break automated verification. For example, VeriF_x does not support traditional loop statements but instead provides higher-order operations (map, filter, etc.) on top of its functional collections. The resulting language is surprisingly expressive given its automated verification capabilities.

The remainder of this section shows how VeriF_x compiles programs to SMT and derives proof obligations that can be discharged automatically by SMT solvers. Afterward, we explain how VeriF_x leverages a specialized theory of arrays to efficiently encode its functional collections. These encodings are key to our approach because they enable fully automated verification of RDTs built atop VeriF_x's functional collections. VeriF_x's encodings significantly differ from related work such as Why3 [26] and Liquid Haskell [75] which encode higher-order operations like map, and filter, recursively which hampers automated verification.

Appendix C.4 exemplifies VeriF_x's compilation rules using a concrete example.

4.1 Core SMT

The semantics of VeriF_x are defined using translation functions from VeriF_x to Core SMT, a reduced version of SMT that suffices to verify VeriF_x programs. Figure 5 defines the syntax of Core SMT. Valid types include integers, strings, booleans, arrays, ADTs $A(\overline{T})$, and user-declared sorts $S(\overline{T})$. Arrays are *total* and map values of the key types to a value of the element type. Arrays can be multidimensional and map several keys to a value.

Core SMT programs consist of one or more statements which can be the declaration of a constant or sort³, assertions, the definition of a function or ADT, or a call to check. Constant declarations take a name and a type. Sort declarations take a name and a non-negative number i representing their arity, i.e. how many type parameters the sort takes.

³ The literature on SMT solvers uses the term “sort” to refer to types and type constructors.

Declared constants and sorts are *uninterpreted* and the SMT solver is free to assign any valid interpretation. Assertions are boolean formulas that constrain the possible interpretations of the program, e.g. `assert age >= 18`.

Function definitions consist of a name f , optional type parameters \overline{X} , formal parameters $\overline{x} : \overline{T}$, a return type T , and a body containing an expression e . Valid expressions include array accesses $e[\tilde{e}]$, array updates $e[\tilde{e}] := e$, anonymous functions, quantified formulas (the full list of expressions is shown in Appendix B). Updating an array returns a modified copy of the array. Note that arrays are total and that anonymous functions define an array. For example, $\lambda(x : \text{int}, y : \text{int}).x + y$ defines an `Array<int, int, int>` that maps two integers to their sum. As arrays are first-class values in SMT, it follows that lambdas are also first-class.

ADT definitions consist of a name A , optional type parameters \overline{X} , and one or more constructors. A constructor has a name K and optionally defines fields with name v and type T . Constructors are called like regular functions and return an instance of the data type.

The decision procedure (`check`) checks the satisfiability of the SMT program. If the program's assertions are satisfiable, `check` returns a concrete model, i.e. an interpretation of the constants and sorts that satisfies the assertions. A property p can be proven by showing that the negation $\neg p$ is unsatisfiable, i.e. that no counterexample exists.

Note that our Core SMT language includes lambdas and polymorphic functions which are not part of SMT-LIB v2.6. Nevertheless, they are described in the preliminary proposal for SMT-LIB v3.0 [35] and Z3 already supports lambdas. For the time being, VeriF_x monomorphizes polymorphic functions when they are compiled to Core SMT. For example, given a polymorphic identity function `id<X> :: X -> X`, VeriF_x creates a monomorphic version `id_int :: int -> int` when encountering a call to `id` with an integer argument.

4.2 Compiling VeriF_x to SMT

Similar to Dafny in [44], we describe the semantics of VeriF_x by means of translation functions that compile VeriF_x to Core SMT. Types are translated by the $\llbracket \cdot \rrbracket_t$ function:

$$\begin{aligned} \llbracket \text{bool} \rrbracket_t &= \text{bool} & \llbracket \text{int} \rrbracket_t &= \text{int} & \llbracket \text{string} \rrbracket_t &= \text{string} \\ \llbracket C(\overline{T}) \rrbracket_t &= C(\llbracket \overline{T} \rrbracket_t) & \llbracket E(\overline{T}) \rrbracket_t &= E(\llbracket \overline{T} \rrbracket_t) & \llbracket \overline{T} \rightarrow P \rrbracket_t &= \text{Array}(\llbracket \overline{T} \rrbracket_t, \llbracket P \rrbracket_t) \end{aligned}$$

Primitive types are translated to the corresponding primitive type in Core SMT. Class types and enumeration types keep the same type name and their type arguments are translated recursively $\llbracket \overline{T} \rrbracket_t$. Functions are encoded as arrays from the argument types to the return type. Trait types do not exist in the compiled SMT program because traits are compiled away by VeriF_x, i.e. only the types of the classes that implement the trait exist in the SMT program.

We now take a look at the translation function $\text{def}[\llbracket \cdot \rrbracket]$ which compiles VeriF_x's main constructs: enumerations, classes, and objects. Enumerations are encoded as ADTs:

$$\text{def}[\llbracket \text{enum } E(\overline{X}) \{ K(\overline{v} : \overline{T}) \} \rrbracket] = \text{adt } E(\overline{X}) \{ K(\overline{v} : \llbracket \overline{T} \rrbracket_t) \}$$

For every enumeration an ADT is constructed with the same name, type parameters, and constructors. The types of the fields are translated recursively.

Classes are encoded as ADTs with one constructor and class methods become functions:

$$\begin{aligned} \text{def}[\llbracket \text{class } C(\overline{X}) (\overline{v} : \overline{T}) \{ \overline{M} \} \text{ extends } I(\overline{P}) \rrbracket] &= \\ &\text{adt } C(\overline{X}) \{ K(\overline{v} : \llbracket \overline{T} \rrbracket_t) \} ; \text{method}[\llbracket C, \overline{X}, M \rrbracket] ; \text{method}[\llbracket C, \overline{X}, M'[\overline{P}/\overline{Y}] \rrbracket] \\ &\text{where } K = \text{str_concat}(C, \text{"_ctor"}) \text{ and } I \text{ is defined as } \text{trait } I(\overline{Y}) \{ \overline{M}' ; \dots \} \\ \text{method}[\llbracket C, \overline{X}, \text{def } m(\overline{Y}) (\overline{x} : \overline{T}) : T_r = e \rrbracket] &= \text{fun } f(\overline{X}, \overline{Y})(\text{this} : C(\overline{X}), \overline{x} : \llbracket \overline{T} \rrbracket_t) : \llbracket T_r \rrbracket_t = \llbracket e \rrbracket \\ &\text{where } f = \text{str_concat}(C, \text{"_"} , m) \end{aligned}$$

The ADT keeps the name of the class and its type parameters, and defines one constructor containing the class' fields. Since the name of the constructor must differ from the ADT's name, the compiler defines a unique name K which is the name of the class followed by “_ctor”. Class methods \bar{M} are compiled to regular functions by function $method[\bar{\cdot}]$. Further, a class inherits all concrete methods \bar{M}' defined by its super trait that are not overridden. This entails substituting the trait's type parameters \bar{Y} by the concrete type arguments \bar{P} defined by the class. As such, traits are compiled away and do not exist in the transpiled SMT program.

For every method, a function is created with a unique name f that is the name of the class followed by an underscore and the name of the method. In the argument list, the body, and the return type of a method, programmers can refer to type parameters of the class and type parameters of the method. Therefore, the compiled SMT function takes both the class' type parameters \bar{X} and the method's type parameters \bar{Y} . Without loss of generality we assume that a method's type parameters do not override the class' type parameters which can be achieved through α -conversion. The method's parameters become parameters of the function. In addition, the function takes an additional parameter *this* referring to the receiver of the method call which should be of the class' type. The types of the parameters and the return type are translated using function $[\bar{\cdot}]_t$. The body of the method must be a well-typed expression. Expressions are translated by the $[\bar{\cdot}]$ function:

$$\begin{array}{lll}
[x] & = & x \\
[\text{val } x : T = e_1 \text{ in } e_2] & = & \text{let } x = [e_1] \text{ in } [e_2] \\
[(x : \bar{T}) \Rightarrow e] & = & \lambda(x : [\bar{T}]_t). [e] \\
[e_1(\bar{e}_2)] & = & [e_1][[\bar{e}_2]] \\
[\text{new } C(\bar{T})(\bar{e})] & = & C'([\bar{T}]_t)([e]) \\
& \text{where } C' = \text{str_concat}(C, \text{"_ctor"}) & \text{and } \bar{P} \cap \bar{T} = \emptyset
\end{array}
\qquad
\begin{array}{ll}
[\text{new } K(\bar{T})(\bar{e})] & = & K([\bar{T}]_t)([e]) \\
[e.v] & = & [e].v \\
[e_1.m(\bar{T})(\bar{e})] & = & m'([\bar{P}]_t, [\bar{T}]_t)([e_1], [e]) \\
& \text{where } \text{typeof}(e_1) = C(\bar{P}) & \\
& \text{and } m' = \text{str_concat}(C, \text{"_"}, m) &
\end{array}$$

Primitive values, variable references, and parameter references remain unchanged in Core SMT. The definition of an immutable variable is translated to a let expression. Anonymous functions remain anonymous functions in Core SMT, the type of the parameters and the body are compiled recursively. Remember that anonymous functions in SMT define (multidimensional) arrays from one or more arguments to the function's return value. Hence, function calls are translated to array accesses. To instantiate a class or ADT, the compiler calls the data type's constructor function. For classes, the constructor's name is the name of the class followed by “_ctor”. To access a field, the compiler translates the expression and accesses the field on the translated expression. To invoke a method m on an object e_1 the compiler calls the corresponding function m' which by convention is the name of the class followed by an underscore and the name of the method. Recall that the function takes both the class' type arguments \bar{T} and the method's type arguments \bar{P} as well as an additional argument e_1 which is the receiver of the call. The complete set of compilation rules for expressions is provided in Appendix C.1 as part of the additional material.

Objects are singletons that can define methods and proofs, and are compiled as follows:

$$\begin{array}{l}
\text{def}[\text{object } O \text{ extends } I(\bar{T}) \{ \bar{M}; \bar{R} \}] = \\
\text{def}[\text{class } O'() \{ \bar{M} \} \text{ extends } I(\bar{T})]; \text{const } O O'; \text{assert } O == O'(); \text{def}[\bar{R}]
\end{array}$$

The object is compiled to a regular class with a fresh name O' . Then, a single instance of that class is created and assigned to a constant named after the object O . The proofs defined by the object are compiled to functions. This translation is the subject of the next section.

4.3 Deriving Proof Obligations

We previously verified a 2PSet CRDT using VeriF_x's CRDT library which internally uses our novel proof construct to define the necessary correctness properties (discussed in Section 5). However, programmers can also define custom proofs, for instance to verify data invariants.

We now explain how proof obligations are derived from user-defined proofs in VeriF_x programs. Proofs are compiled to regular functions without arguments. While, the name and type parameters remain unchanged, the body of the proof is compiled and becomes the function's body. Proofs always return a boolean since the body is a logical formula whose satisfiability must be checked.

$$\text{def } \llbracket \text{proof } p \langle \bar{X} \rangle \{ e \} \rrbracket = \text{fun } p \langle \bar{X} \rangle () : \text{bool} = \llbracket e \rrbracket$$

To check if the property described by a proof holds, the negation of the proof must be unsatisfiable – if no counterexample exists it constitutes a proof that the property is correct. A (polymorphic) proof called p with zero or more type parameters i is checked as follows:

$$\text{prove}(p, i) = \text{sort } S_1 \ 0 ; \dots ; \text{sort } S_i \ 0 ; \text{assert } \neg p \langle S_1, \dots, S_i \rangle () ; \text{check}() == \text{UNSAT}$$

For every type parameter, an *uninterpreted* sort is declared. Then, the proof function is called with those sorts as type arguments and we check that the negation is unsatisfiable. If the negation is unsatisfiable, the (polymorphic) proof holds for all possible instantiations of its type parameters. The underlying SMT solver can generate an actual proof which could be reconstructed by proof assistants as shown by Böhme et al. [15], Böhme and Weber [16].

4.4 Encoding Functional Collections Efficiently in SMT

Some IVLs feature collections with rich APIs (e.g. Why3 [26]) but encode operations on these collections recursively. Traditional SMT solvers fail to verify recursive definitions automatically because they require inductive proofs, which is beyond the capabilities of most solvers. However, many SMT solvers support specialized array theories. A key insight of this paper consists of efficiently encoding the collections and their operations using the Combinatory Array Logic (CAL) [23] which is decidable. As a result, VeriF_x can automatically verify RDTs that are built by arbitrary compositions of functional collections. Next, we describe the encoding of sets using this array logic, while maps are described in Appendix C.3.

Set Encoding. Sets are encoded as arrays from the element type to a boolean type that indicates whether the element is in the set:

$$\llbracket \text{Set } \langle T \rangle \rrbracket_t = \text{Array}(\llbracket T \rrbracket_t, \text{bool})$$

An empty set corresponds to an array containing false for every element. We can create such an array by defining a lambda that ignores its argument and always returns false:

$$\llbracket \text{new Set } \langle T \rangle () \rrbracket = \lambda(x : \llbracket T \rrbracket_t). \text{false}$$

Operations on sets are compiled as follows:

$$\begin{aligned} \llbracket e_1.add(e_2) \rrbracket &= \llbracket e_1 \rrbracket[\llbracket e_2 \rrbracket] := \text{true} & \llbracket e_1.remove(e_2) \rrbracket &= \llbracket e_1 \rrbracket[\llbracket e_2 \rrbracket] := \text{false} \\ \llbracket e_1.filter(e_2) \rrbracket &= \lambda(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \wedge \llbracket e_2 \rrbracket[x] & \llbracket e_1.contains(e_2) \rrbracket &= \llbracket e_1 \rrbracket[\llbracket e_2 \rrbracket] \\ \text{where } \text{typeof}(e_1) &= \text{Set} \langle T \rangle \wedge \text{typeof}(e_2) = T \rightarrow \text{bool} & & \\ \llbracket e_1.map(e_2) \rrbracket &= \lambda(y : \llbracket P \rrbracket_t). \exists(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \wedge \llbracket e_2 \rrbracket[x] = y & & \\ \text{where } \text{typeof}(e_1) &= \text{Set} \langle T \rangle \wedge \text{typeof}(e_2) = T \rightarrow P & & \end{aligned}$$

An element e_2 is added to a set e_1 by setting the entry for e_2 in the array that results from transforming e_1 to true. Similarly, an element is removed by changing its entry in the array to false. An element is in the set if its entry is true. A set e_1 containing elements of type

T can be filtered such that only the elements that fulfil a given predicate $e_2 : T \rightarrow \text{bool}$ are retained. Calls to *filter* are compiled to a lambda that defines a set (i.e. an array from elements to booleans) containing only the elements x that are in the original set e_1 (i.e. $\llbracket e_1 \rrbracket [x]$) and fulfil predicate e_2 (i.e. $\llbracket e_2 \rrbracket [x]$). Similarly, a function $e_2 : T \rightarrow P$ can be mapped over a set e_1 of T s, yielding a set of P s. Calls to *map* are compiled to a lambda that defines a set containing elements y of type $\llbracket P \rrbracket_t$ such that an element x exists that is in the original set e_1 (i.e. $\llbracket e_1 \rrbracket [x]$) and maps to y (i.e. $\llbracket e_2 \rrbracket [x] = y$). The remaining methods are described in Appendix C.2 as part of the additional material.

5 Implementing and Verifying Replicated Data Types

VeriF_x aims to simplify the development of correct RDTs by integrating automated verification capabilities in the language. Based on our experience implementing RDTs, we noticed that RDTs need to fulfill specific correctness properties that are well-defined for each RDT family. Therefore, VeriF_x features built-in libraries for the development and automated verification of two well-known RDT families: CRDTs [68] and OT [25]. These libraries are written in VeriF_x and define proofs that encode the necessary correctness properties such that programmers do not need to redefine these proofs for every RDT they implement.

This section discusses the aforementioned libraries. For each library, we formally define the correctness properties that must be verified for that specific RDT family. Section 5.1 describes the implementation of a general execution model for CRDTs and its verification library in VeriF_x. Next, Section 5.2 presents a library for implementing RDTs using OT and verifying the transformation functions. VeriF_x is not limited to these families of RDTs; programmers can build custom libraries for implementing and verifying other abstractions or families of RDTs. Last, Section 5.3 explains how to encode common assumptions such as causal delivery in VeriF_x since the CRDT and OT libraries do not make specific assumptions.

5.1 CRDT Library

CRDTs guarantee *strong eventual consistency* (SEC), a consistency model that strengthens eventual consistency with the *strong convergence* property which requires replicas that received the same updates, possibly in a different order, to be in the same state. VeriF_x's CRDT library supports all CRDT families: state-based [68], delta state-based [2], op-based [68], and pure op-based CRDTs [8]. The remainder explains how our library supports each family.

5.1.1 State-based CRDTs

State-based CRDTs (CvRDTs for short) periodically broadcast their state to all replicas and merge incoming states by computing the least upper bound (LUB) of the incoming state and their own state. Shapiro et al. [68] showed that CvRDTs converge if the merge function \sqcup_v is idempotent, commutative, and associative. We define these properties based on their work:

Idempotent: $\forall x \in \Sigma : \text{reachable}(x) \implies x \equiv x \sqcup_v x$

Commutative: $\forall x, y \in \Sigma : \text{reachable}(x) \wedge \text{reachable}(y) \wedge \text{compatible}(x, y) \implies (x \sqcup_v y \equiv y \sqcup_v x) \wedge \text{reachable}(x \sqcup_v y)$

Associative: $\forall x, y, z \in \Sigma : \text{reachable}(x) \wedge \text{reachable}(y) \wedge \text{reachable}(z) \wedge \text{compatible}(x, y) \wedge \text{compatible}(x, z) \wedge \text{compatible}(y, z) \implies ((x \sqcup_v y) \sqcup_v z \equiv x \sqcup_v (y \sqcup_v z)) \wedge \text{reachable}((x \sqcup_v y) \sqcup_v z)$

■ **Listing 4** Trait for the implementation of CvRDTs in VeriF_x.

```

1 trait CvRDT[T <: CvRDT[T]] {
2   def merge(that: T): T
3   def compare(that: T): Boolean
4   def reachable(): Boolean = true
5   def compatible(that: T): Boolean =
6     true
7   def equals(that: T): Boolean = {
8     this.asInstanceOf[T].compare(that)
9     &&
10    that.compare(this.asInstanceOf[T])
11  }

```

■ **Listing 5** Trait for the verification of CvRDTs in VeriF_x. The arrow function =>: implements logical implication.

```

1 trait CvRDTProof[T <: CvRDT[T]] {
2   proof mergeIdempotent {
3     forall (x: T) { x.reachable() => x.merge(x).equals(x) } }
4   proof mergeCommutative {
5     forall (x: T, y: T) {
6       (x.reachable() && y.reachable() && x.compatible(y)) =>
7       (x.merge(y).equals(y.merge(x)) &&
8        x.merge(y).reachable())}
9   proof mergeAssociative {
10    forall (x: T, y: T, z: T) {
11      (x.reachable() && y.reachable() && z.reachable() &&
12       x.compatible(y) && x.compatible(z) && y.compatible(z))
13      => (x.merge(y).merge(z).equals(x.merge(y.merge(z))) &&
14         x.merge(y).merge(z).reachable()) } }
15  proof equalityCheck {
16    forall (x: T, y: T) { x.equals(y) == (x == y) } } }

```

Σ denotes the set of all states. A state is *reachable* if it can be reached starting from the initial state and applying only supported operations. Two states are *compatible* if they represent different replicas of the same CRDT object⁴. As explained in Section 2.1, Shapiro et al. [68] define state equivalence in terms of \leq_v on the lattice: $S \leq_v T \wedge T \leq_v S \implies S \equiv T$.

VeriF_x's CRDT library provides traits for the implementation and verification of CvRDTs, shown in Listings 4 and 5 respectively. Listing 4 shows the `CvRDT` trait that was used in Listing 1 to implement the `TwoPSet` CRDT. Every state-based CRDT that extends the `CvRDT` trait must provide a type argument which is the actual type of the CRDT and provide an implementation for the `merge` and `compare` methods. By default, all states are considered reachable and compatible, and state equivalence is defined in terms of `compare`. These methods can be overridden by the concrete CRDT that implements the trait.

Listing 5 shows the `CvRDTProof` trait used to verify CvRDT implementations. This trait defines one type parameter `T` that must be a CvRDT type and defines proofs to check that its merge function adheres to the aforementioned properties (i.e. is idempotent, commutative, and associative). It also defines an additional proof, `equalityCheck`, that checks that the notion of state equivalence that, by default, relies on structural equality (i.e., the default implementation of the `equals` method computes structural equality). However, programmers can override the `equals` method to use another notion of state equivalence if needed.

Objects can extend the `CvRDTProof` trait to inherit automated correctness proofs for the given CRDT type. Note that the trait's type parameter `T` expects a concrete CvRDT type (e.g. `PNCOUNTER`) and will not work for polymorphic CvRDTs (e.g. `ORSet`) because those are type constructors. Instead, the CRDT library provides additional `CvRDTProof1`, `CvRDTProof2`, and `CvRDTProof3` traits to verify polymorphic CvRDTs that expect 1, 2, or 3 type arguments respectively. For example, the `TwoPSet[V]` from Section 2 is polymorphic in the type of values it stores; the `TwoPSetProof` object thus extended the `CvRDTProof1` trait because the `TwoPSet` expects one type argument.

5.1.2 Delta state-based CRDTs

Delta state-based CRDTs are a family of state-based CRDTs that exchange only the changes to the state (called deltas) instead of the full state in order to reduce the amount of data that is sent. Mutator operations return a delta which is joined into the replica's local state, propagated to the other replicas, and eventually joined into the state of all replicas.

⁴ The `compatible` predicate can be used to encode certain assumptions. For example, replicas have unique identifiers which enables them to generate unique tags.

VeriF_x's CRDT library provides a `DeltaCRDT` trait that specializes the `CvRDT` trait and can be used to implement delta state-based CRDTs. When extending `DeltaCRDT` traits, programmers must implement a `merge` method that joins delta states into the local state.

To verify delta state-based CRDTs, programmers can reuse the `CvRDTProof` trait since delta state-based CRDTs are essentially state-based CRDTs. As shown in Listing 5, the `CvRDTProof` trait verifies that the `merge` is idempotent, commutative, and associative **for all** valid states. This valid states contain all valid delta states as they are a subset of the full state.

5.1.3 Op-based CRDTs

Op-based CRDTs (`CmRDTs` for short) execute update operations in two phases, called *prepare* and *effect*. The prepare phase executes locally at the source replica (only if its source precondition holds) and prepares a message to be broadcast⁵ to all replicas (including itself). The effect phase applies such incoming messages and updates the state (only if its downstream precondition holds, otherwise the message is ignored).

Shapiro et al. [68] and Gomes et al. [27] have shown that `CmRDTs` guarantee SEC if all concurrent operations commute. Hence, for any `CmRDT` it suffices to show that all pairs of concurrent operations commute. Formally, for any operation o_1 that is enabled by some reachable replica state s_1 (i.e. o_1 's source precondition holds in s_1) and any operation o_2 that is enabled by some reachable replica state s_2 , if these operations can be concurrent, and s_1 , s_2 , and s_3 are compatible replica states, then we must show that on any reachable replica state s_3 the operations commute and the intermediate and resulting states are reachable:

$$\begin{aligned} & \forall s_1, s_2, s_3 \in \Sigma, \forall o_1, o_2 \in \Sigma \rightarrow \Sigma : \text{reachable}(s_1) \wedge \text{reachable}(s_2) \wedge \text{reachable}(s_3) \wedge \\ & \quad \text{enabledSrc}(o_1, s_1) \wedge \text{enabledSrc}(o_2, s_2) \wedge \text{canConcur}(o_1, o_2) \wedge \\ & \quad \text{compatible}(s_1, s_2) \wedge \text{compatible}(s_1, s_3) \wedge \text{compatible}(s_2, s_3) \\ & \implies o_2 \cdot o_1 \cdot s_3 \equiv o_1 \cdot o_2 \cdot s_3 \wedge \text{reachable}(o_1 \cdot s_3) \wedge \text{reachable}(o_2 \cdot s_3) \wedge \text{reachable}(o_1 \cdot o_2 \cdot s_3) \end{aligned}$$

We use the notation $o \cdot s$ to denote the application of an operation o on state s if its downstream precondition holds, otherwise, it returns the state unchanged.

Listing 6 shows the `CmRDT` trait that must be extended by op-based CRDTs with concrete type arguments for the supported operations, exchanged messages, and the CRDT type itself. A CRDT that extends the `CmRDT` trait must implement the `prepare` and `effect` methods. The `tryEffect` method has a default implementation that applies the operation if its downstream precondition holds, otherwise, it returns the state unchanged. By default, we assume all states are reachable, all operations are enabled at the source and downstream, all operations can occur concurrently, and all states are compatible. For most `CmRDTs` these settings do not need to be altered but some `CmRDTs` have other assumptions which can be encoded by overriding the appropriate method. For example, in an OR-Set [67] it is not possible to delete tags added concurrently; this can be encoded by overriding `canConcur`.

Similar to state-based CRDTs, our CRDT library provides a `CmRDTProof` trait and several versions to verify op-based CRDTs. These traits define a general proof of correctness that checks that all operations commute based on the previously described formula.

5.1.4 Pure op-based CRDTs

Pure op-based CRDTs are a family of op-based CRDTs that exchange only the operations instead of data-type specific messages. The effect phase stores incoming operations in a partially ordered log of (concurrent) operations. Queries are computed against the log and

⁵ While some `CmRDT` do not require causal delivery, the overall model assumes reliable causal broadcast.

■ **Listing 6** Polymorphic CmRDT trait to implement op-based CRDTs in VeriFx.

```

1  trait CmRDT[Op, Msg, T <: CmRDT[Op, Msg, T]] {
2    def prepare(op: Op): Msg
3    def effect(msg: Msg): T
4    def tryEffect(msg: Msg): T = if (this.enabledDown(msg)) this.effect(msg) else this.asInstanceOf[T]
5    def reachable(): Boolean = true // by default all states are considered reachable
6    def canConcur(x: Msg, y: Msg): Boolean = true // all ops can occur concurrently
7    def compatible(that: T): Boolean = true // all states are compatible
8    def enabledSrc(op: Op): Boolean = true // no source preconditions by default
9    def enabledDown(msg: Msg): Boolean = true // no downstream preconditions by default
10   def equals(that: T): Boolean = this == that
11 }

```

operations do not need to commute. Data-type specific redundancy relations dictate which operations to store in the log and when to remove operations from the log. VeriFx’s CRDT library provides a `PureOpBasedCRDT` trait which is a specialization of the `CmRDT` trait for implementing pure op-based CRDTs. The implementing CRDT inherits the prepare and effect phase (the same for all pure op-based CRDTs) and only needs to implement the redundancy relations. Since pure op-based CRDTs are essentially operation-based CRDTs, programmers can reuse the `CmRDTProof` traits to verify pure op-based CRDT implementations.

5.2 OT Library

The Operational Transformation (OT) [25] approach applies operations locally and propagates them asynchronously to the other replicas. Incoming operations are transformed against previously executed concurrent operations such that the modified operation preserves the intended effect. Operations are functions from state to state: $Op : \Sigma \rightarrow \Sigma$ and are transformed using a transformation function $T : Op \times Op \rightarrow Op$. Thus, $T(o_1, o_2)$ denotes the operation that results from transforming o_1 against a previously executed concurrent operation o_2 . Suleiman et al. [70] and Sun et al. [72] proved that replicas eventually converge if the transformation function satisfies two properties: TP_1 and TP_2 . Property TP_1 states that any two enabled concurrent operations o_i and o_j must commute after transforming them:

$$\forall o_i, o_j \in Op, \forall s \in \Sigma : \text{enabled}(o_i, s) \wedge \text{enabled}(o_j, s) \wedge \text{canConcur}(o_i, o_j) \\ \implies T(o_j, o_i)(o_i(s)) = T(o_i, o_j)(o_j(s))$$

Property TP_2 states that given three enabled concurrent operations o_i , o_j , and o_k , the transformation of o_k does not depend on the order of the transformation of operations o_i and o_j :

$$\forall o_i, o_j, o_k \in Op, \forall s \in \Sigma : \text{enabled}(o_i, s) \wedge \text{enabled}(o_j, s) \wedge \text{enabled}(o_k, s) \wedge \text{canConcur}(o_i, o_j) \wedge \\ \text{canConcur}(o_j, o_k) \wedge \text{canConcur}(o_i, o_k) \implies T(T(o_k, o_i), T(o_j, o_i)) = T(T(o_k, o_j), T(o_i, o_j))$$

Note that properties TP_1 and TP_2 only need to hold for states in which the operations can be generated, represented by the relation $\text{enabled} : Op \times \Sigma \rightarrow \mathbb{B}$, and only if the two operations can occur concurrently, represented by the relation $\text{canConcur} : Op \times Op \rightarrow \mathbb{B}$.

VeriFx provides a library for implementing and verifying RDTs that use operational transformations. Programmers can build custom RDTs by extending the `OT` trait shown in Listing 7. Every RDT that extends the `OT` trait must provide concrete type arguments for the state and operations, and implement the `transform` and `apply` methods. The `transform` method transforms an incoming operation against a previously executed concurrent operation. The `apply` method applies an operation on the state. By extending this trait, the RDT inherits proofs for TP_1 and TP_2 . By default, these proofs assume that operations are always enabled and that all operations can occur concurrently. If this is not the case, the RDT can override the `enabled` and `canConcur` methods respectively.

■ **Listing 7** Polymorphic OT trait to implement and verify RDTs using operational transformation.

```

1 trait OT[State, Op] {
2   def transform(x: Op, y: Op): Op
3   def apply(state: State, op: Op): State
4   def enabled(op: Op, state: State): Boolean = true
5   def canConcur(x: Op, y: Op): Boolean = true
6   proof TP1 {
7     forall (opI: Op, opJ: Op, st: State) {
8       (this.enabled(opI, st) && this.enabled(opJ, st) && this.canConcur(opI, opJ)) => {
9         this.apply(this.apply(st, opI), this.transform(opJ, opI)) ==
10        this.apply(this.apply(st, opJ), this.transform(opI, opJ)) } } }
11  proof TP2 {
12    forall (opI: Op, opJ: Op, opK: Op, st: State) {
13      (this.enabled(opI, st) && this.enabled(opJ, st) && this.enabled(opK, st) &&
14       this.canConcur(opI, opJ) && this.canConcur(opJ, opK) && this.canConcur(opI, opK)) => {
15        this.transform(this.transform(opK, opI), this.transform(opJ, opI)) ==
16        this.transform(this.transform(opK, opJ), this.transform(opI, opJ)) } } } }

```

Although VeriFx supports the general execution model of OT, most transformation functions described by the literature were specifically designed for collaborative text editing. They model text documents as a sequence of characters and operations insert or delete characters at a given position in the document. Every paper thus describes four transformations functions, one for every pair of operations: insert-insert, insert-delete, delete-insert, delete-delete.

Likewise, VeriFx’s OT library provides a `ListOT` trait that models the state as a list of values and supports insertions and deletions. RDTs extending the `ListOT` trait need to implement four methods (`Tii`, `Tid`, `Tdi`, `Tdd`) corresponding to the transformation functions for transforming insertions against insertions (`Tii`), insertions against deletions (`Tid`), deletions against insertions (`Tdi`), and deletions against deletions (`Tdd`). The trait provides a default implementation of `transform` that dispatches to the corresponding transformation function based on the type of operations, and a default implementation of `apply` that inserts or deletes a value from the underlying list.

5.3 Encoding RDT-Specific Assumptions

Some RDTs (most notably op-based CRDTs) assume causal delivery of operations but VeriFx and its CRDT and OT libraries do not make any assumptions. In VeriFx, assumptions must either be guaranteed by the RDT’s implementation or be explicitly encoded in the proofs.

We now show, using the OR-Set CRDT [67], how to encode RDT-specific assumptions. The OR-Set CRDT assumes that 1) replicas can generate globally unique tags, and 2) add and remove operations of the same element are delivered in causal order. These assumptions imply that replicas cannot add a tag and concurrently remove the same tag. The first assumption can be guaranteed by the RDT implementation if every replica has a unique ID that is combined with a local counter that increases monotonically to generate unique tags. The latter assumption about causal delivery can be explicitly encoded in the proof. One could also model the underlying causal communication protocols in VeriFx to remove this assumption.

Listing 8 shows an excerpt from the implementation of the OR-Set CRDT. It overrides the `compatible` predicate (Line 7) to encode the fact that replicas have unique IDs, and overrides the `canConcur` predicate (Line 8 to 16) such that the proof does not consider `add` and `remove` operations if the tag generated by `add` is contained in the set of tags that are removed (because causal delivery precludes `remove` from having observed that tag). This example shows how to encode specific assumptions but, in practice, many RDT implementations do not require any assumptions. Only 7 out of the 51 verified CRDTs (cf. Section 6.1) required assumptions, all of which are related to causal delivery and logical timestamps.

■ **Listing 8** Excerpt from the implementation of the OR-Set CRDT [67].

```

1 class Tag[ID](replica: ID, counter: Int)
2 enum SetOp[V, ID] { Add(e: V) | Remove(e: V) }
3 enum SetMsg[V, ID] { AddMsg(e: V, tag: Tag[ID]) | RemoveMsg(e: V, tags: Set[Tag[ID]]) }
4 class ORSet[V, ID](id: ID, counter: Int, elements: Map[V, Set[Tag[ID]]])
5   extends CmRDT[SetOp[V, ID], SetMsg[V, ID], ORSet[V, ID]] {
6   // ...
7   override def compatible(that: ORSet[V, ID]) = this.id != that.id
8   override def canConcur(x: SetMsg[V, ID], y: SetMsg[V, ID]) = x match {
9     case AddMsg(_, tag) => y match {
10      case AddMsg(_, _) => true
11      case RemoveMsg(_, tags) => !tags.contains(tag) // tag cannot be in tags because of causal delivery
12    }
13    case RemoveMsg(_, tags) => y match {
14      case AddMsg(_, tag) => !tags.contains(tag) // tag cannot be in tags because of causal delivery
15      case RemoveMsg(_, _) => true
16  } } }

```

6 Evaluation

We now evaluate the applicability of VeriF_x to implement and verify RDTs. Our evaluation is twofold. First, we implement and verify numerous CRDTs taken from literature as well as some new variants⁶. Also, we verify well-known OT functions and some unpublished designs.

All experiments reported were conducted on AWS using an m5.xlarge VM with 4 virtual CPUs and 16 GiB of RAM. All benchmarks are implemented using JMH [61], a benchmarking library for the JVM. We configured JMH to execute 20 warmup iterations followed by 20 measurement iterations for every benchmark. To avoid run-to-run variance JMH repeats every benchmark in 3 fresh JVM forks, yielding a total of 60 samples per benchmark.

We do not conduct a performance evaluation for the transpiled RDT implementations as the transpilation merely changes the syntax to Scala or JavaScript but does not modify the RDT’s design. Thus, the transpilation step does not affect the RDT’s performance.

6.1 Verifying Conflict-free Replicated Data Types

We implemented and verified an extensive portfolio comprising 51 CRDTs, coming from literature [2, 8, 14, 40, 66, 67], open source projects [9], and industrial databases [1, 12, 52]. To the best of our knowledge, we are the first to mechanically verify all CRDTs from Shapiro et al. [67], all delta state-based CRDTs from Almeida et al. [2], all pure op-based CRDTs from Baquero et al. [8], and the map CRDT from Kleppmann [40].

Table 1 summarizes the verification results, including the average verification time and code size of each CRDT. When applicable, we mention which CRDTs are used in industrial databases. VeriF_x was able to verify all implemented CRDTs except the Replicated Growable Array (RGA) [67] due to the recursive nature of the insertion algorithm (cf. Section 7). We found three issues: 1) the Two-Phase Set CRDT (described in Section 2) converges but is not functionally correct, 2) the original Map CRDT proposed by Kleppmann [40] diverges as VeriF_x found the same counterexample as described in their technical report, and 3) the Molli, Weiss, Skaf (MWS) Set is incomplete. We now describe the implementation and verification of the Map CRDTs from [40], while Appendix E discusses the MWS Set.

⁶ All implementations and proofs are provided as supplementary material in this submission.

■ **Table 1** Verification results for CRDTs implemented and verified in VeriF_x. S = state-based, D = delta state-based, O = op-based, P = pure op-based CRDT. ⊙ = timeout, @ = adaptation of an existing CRDT, ⊕ = incomplete definition. The database column includes databases that are known to use these CRDTs. Some delta state-based CRDTs use a dot kernel abstraction that is not counted in the LoC, this is indicated in the LoC column with an asterisk.

CRDT	Type	LoC	Correct	Time	Database	Source
Counter	O	17	✓	3.2 s	AntidoteDB	Shapiro et al. [67]
Grow-Only Counter	S	27	✓	4.3 s		Shapiro et al. [67]
Grow-Only Counter	D	27	✓	4.4 s	Akka	Almeida et al. [2]
Dynamic Grow-Only Counter	S	27	✓	4.4 s	Riak	@ Shapiro et al. [67]
Positive-Negative Counter	S	12	✓	5.9 s		Shapiro et al. [67]
Positive-Negative Counter	D	17	✓	6.8 s	Akka	Almeida et al. [2]
Dynamic Positive-Negative Counter	S	17	✓	9.3 s	Riak	@ Shapiro et al. [67]
Lex Counter	D	46	✓	4.7 s	Cassandra	Baquero et al. [9]
Causal Counter	D	28*	✓	6.7 s	Riak	Baquero et al. [9]
Enable-Wins Flag	P	18	✓	4.0 s		Baquero et al. [8]
Enable-Wins Flag	D	14*	✓	5.7 s	Riak	Baquero et al. [9]
Enable-Wins Flag	O	44	✓	3.6 s	AntidoteDB	AntidoteDB [4]
Disable-Wins Flag	P	20	✓	3.9 s		Baquero et al. [8]
Disable-Wins Flag	D	14*	✓	5.8 s	Riak	Baquero et al. [9]
Disable-Wins Flag	O	50	✓	3.8 s	AntidoteDB	AntidoteDB [3]
Multi-Value Register	S	63	✓	8.8 s		Shapiro et al. [67]
Multi-Value Register	D	12*	✓	7.1 s		Almeida et al. [2]
Multi-Value Register	P	18	✓	4.1 s		Baquero et al. [8]
Last-Writer-Wins Register	S	16	✓	5.3 s	Riak	Shapiro et al. [67]
Last-Writer-Wins Register	O	38	✓	4.4 s		Shapiro et al. [67]
Grow-Only Set	O	17	✓	3.9 s	AntidoteDB	Shapiro et al. [67]
Grow-Only Set	S	8	✓	5.3 s	Riak	Shapiro et al. [67]
Grow-Only Set	D	9	✓	3.9 s		Baquero et al. [9]
Two-Phase Set	O	27	✓	4.4 s		Shapiro et al. [67]
Two-Phase Set	S	16	✗	6.3 s		Shapiro et al. [67]
Two-Phase Set	D	25	✓	4.5 s		Baquero et al. [9]
Unique Set	O	39	✓	4.4 s		Shapiro et al. [67]
Add-Wins Set	P	28	✓	4.3 s		Baquero et al. [8]
Remove-Wins Set	P	42	✓	4.5 s		Baquero et al. [8]
Last-Writer-Wins Set	S	36	✓	6.6 s		Shapiro et al. [67]
Remove-Wins Last-Writer-Wins Set	D	28	✓	4.8 s		Baquero et al. [9]
Positive-Negative Set	S	36	✓	9.6 s		Shapiro et al. [67]
Observed-Removed Set	O	75	✓	6.2 s	AntidoteDB	Shapiro et al. [67]
Observed-Removed Set	S	34	✓	7.6 s		Shapiro [66]
Optimized OR Set	S	78	✓	30.2 s	Riak	Bieniusa et al. [14]
Add-Wins OR Set	D	28	✓	6.5 s		Almeida et al. [2]
Optimized Add-Wins OR-Set	D	16*	✓	7.3 s		Almeida et al. [2]
Optimized Remove-Wins OR-Set	D	27*	✓	8.5 s		Baquero et al. [9]
Molli, Weiss, Skaf (MWS) Set	O	45	✓	4.7 s		⊕ Shapiro et al. [67]
Grow-Only Map	S	32	✓	9.1 s		new data type
Buggy Map	O	87	✗	65.2 s		Kleppmann [40]
Corrected Map	O	101	✓	49.4 s		Kleppmann [40]
2P2P Graph	O	58	✓	7.8 s		Shapiro et al. [67]
2P2P Graph	S	41	✓	10.7 s		@ Shapiro et al. [67]
Add-Only Directed Acyclic Graph	O	42	✓	4.7 s		Shapiro et al. [67]
Add-Only Directed Acyclic Graph	S	30	✓	8.7 s		@ Shapiro et al. [67]
Add-Remove Partial Order	O	61	✓	10.4 s		Shapiro et al. [67]
Add-Remove Partial Order	S	49	✓	13.2 s		@ Shapiro et al. [67]
Replicated Growable Array	O	156	⊙	/		Shapiro et al. [67]
Continuous Sequence	O	108	✓	9.2 s		@ Shapiro et al. [67]
Continuous Sequence	S	53	✓	11.4 s		@ Shapiro et al. [67]

6.1.1 Map CRDTs

Kleppmann [40] describes the implementation of a Map CRDT which he believed to be “obviously correct” only to find out it contains a bug that causes divergence after spending hours trying to verify it. He then tweeted the buggy pseudo code of the Map CRDT and challenged his 29400 followers (mainly software engineers) to find the bug. Only one person managed to manually identify the bug and one other person came close (at the time, both were Ph.D. students specialized in RDTs). Kleppmann later tweeted a variation on the algorithm: “Here is a variant of the algorithm that is correct (I believe)”.

We used VeriFx to implement and *automatically* verify both the buggy Map CRDT and the corrected Map CRDT, which had not been formally verified. The full implementation and verification of the buggy map CRDT is explained in Appendix D. We now present the key takeaways from our experience implementing and verifying these map CRDTs.

Implementation. The implementation of the map CRDTs mainly consisted of translating the mathematical specifications to VeriFx. We introduced slight changes to the design to improve efficiency. For example, the specification keeps a set of triples where each triple holds a key, a value, and a timestamp. Since every key appears at most in one triple, our implementation uses a dictionary to efficiently map keys to their value and timestamp.

Verification. After implementing the buggy map CRDT, we proceeded to its automated verification but VeriFx generated invalid counterexamples. For instance, one in which two distinct replicas generated the same timestamp. This is not possible because the design assumes that replicas have unique IDs and combines them with Lamport clocks [43] to generate unique timestamps. However, VeriFx does not know this assumption nor does it know the relation between a replica’s clock and the values it observed. In practice, other CRDTs make similar *implicit* assumptions which is the reason they are complex and difficult to get right. VeriFx helped us to explicitly encode all assumptions as it kept returning invalid counterexamples which helped us find and formulate the missing assumptions. Listing 11 in Appendix D.3 shows the encoding of these assumptions.

Counterexample. After explicitly defining all assumptions, VeriFx found a valid counterexample for the buggy map CRDT that is equivalent to the one found manually by Nair [40]. It consists of a corner case in which the `Put` and `Delete` operations do not commute and thus may cause replicas to diverge. We detail the counterexample in Appendix D.3.

Corrected Map CRDT. After finding the counterexample for the buggy map CRDT, we also verified the corrected map CRDT from Kleppmann [40]. This did not require additional efforts since we already distilled all assumptions for the buggy map CRDT. VeriFx automatically proved that the corrected design indeed guarantees convergence, which to the best of our knowledge, is the first mechanical proof of correctness for this CRDT.

As shown in Table 1, the verification times for the buggy and corrected map CRDTs are slightly higher compared to the other CRDTs we verified, but are still very fast for a fully automated verification approach. The higher times come from the fact that these CRDTs are too complex to directly prove convergence of all operation pairs. Hence, we use a subproof for every operation pair. The total verification time is the sum of the times of the subproofs.

6.1.2 Conclusion

Based on Table 1, we conclude that VeriFx is suited to verify CRDT implementations since all were verified mechanically and fully automatically in a matter of seconds. To the best of our knowledge, this is the most extensive portfolio of verified RDTs to date. It is representative of real-world use cases as it includes several CRDTs used in industrial databases.

■ **Table 2** Verification results of OT functions in VeriF_x.

Transformation Function	LoC	Props		Time	
		TP_1	TP_2	TP_1	TP_2
Ellis and Gibbs [25]	84	✗	✗	115 s	29 s
Ressel et al. [64]	78	✓	✗	68 s	30 s
Sun et al. [72]	68	✗	✗	321 s	13 s
Suleiman et al. [69]	85	✗	✗	34 s	40 s
Imine et al. [34]	83	✓	✗	61 s	17 s
Register _{v₁} [33]	6	✗	✓	3 s	3 s
Register _{v₂} [33]	6	✓	✗	3 s	3 s
Register _{v₃} [33]	7	✓	✓	3 s	3 s
Stack [33]	47	✗	✓	5 s	5 s

■ **Listing 9** Excerpt from the implementation of Imine et al. [34]’s functions.

```

1  enum Op { Ins(p: Int, ip: Int, c: Int) |
      Del(p: Int) | Id() }
2  object Imine extends ListOT[Int, Op] {
3    def Tii(x: Ins, y: Ins) = {
4      val p1 = x.p; val ip1 = x.ip; val c1 = x.c
5      val p2 = y.p; val ip2 = y.ip; val c2 = y.c
6      if (p1 < p2) x
7      else if (p1 > p2) new Ins(p1 + 1, ip1, c1)
8      else if (ip1 < ip2) x
9      else if (ip1 > ip2) new Ins(p1+1, ip1, c1)
10     else if (c1 < c2) x
11     else if (c1 > c2) new Ins(p1+1, ip1, c1)
12     else new Id() }
13   def Tid(x: Ins, y: Del) =
14     if (x.p > y.p) new Ins(x.p - 1, x.ip, x.c)
15     else x
16   def Tdi(x: Del, y: Ins) =
17     if (x.p < y.p) x else new Del(x.p + 1)
18   def Tdd(x: Del, y: Del) = if (x.p < y.p) x
19     else if (x.p > y.p) new Del(x.p - 1)
20     else new Id() }

```

Overall, the main challenge to building such an extensive portfolio consisted of finding and encoding the correct assumptions. Those assumptions were usually gradually discovered as VeriF_x returned counterexamples that cannot occur in practice, which indicates that one or more assumptions are missing. In particular, counterexamples were crucial for verifying the Map CRDTs from [40]. These are the designs that took the longest to implement and verify and they required an afternoon of work.

6.2 Verifying Operational Transformation

We now show that VeriF_x is general enough to verify other distributed abstractions such as Operational Transformation (OT). We implemented all transformation functions for collaborative text editing defined by Imine et al. [34] and verified TP_1 and TP_2 in VeriF_x.

Table 2 summarizes the verification results. For each transformation function, the table shows the code size, whether or not it satisfies TP_1 and TP_2 , and the average verification time. As shown in the table, the functions proposed by Ellis and Gibbs [25], Sun et al. [72], and Suleiman et al. [69] do not satisfy TP_1 nor TP_2 . Ressel et al. [64]’s functions satisfy TP_1 but not TP_2 . These results confirm prior findings by Imine et al. [34]. VeriF_x also found that the functions proposed by Imine et al. [34] do not satisfy TP_2 , which confirms the findings of Li and Li [49] and Oster et al. [62]. That same counterexample also invalidates the transformation functions of Suleiman for TP_2 . Imine et al. [34] wrongly proved Suleiman’s functions [69] correct, but VeriF_x found counterexamples for both properties (the counterexample for TP_1 was manually found in [63]). We believe that the specification defined in Imine et al. [34] may have missed those counterexamples due to a wrong encoding of assumptions. Finally, in a private communication, Imine [33] asked us to verify (unpublished) OT designs for replicated registers and stacks. Out of the three register designs verified in VeriF_x, only one is correct for both TP_1 and TP_2 . Regarding the stack design, it guarantees TP_2 but not TP_1 . VeriF_x provided meaningful counterexamples for each incorrect design.

To exemplify our approach to verifying OT, we now describe the implementation and verification of Imine et al. [34]’s transformation functions in VeriF_x, which are shown in Listing 9. The enumeration `Op` on Line 1 defines the three supported operations:

- `Ins(p, ip, c)` represents the insertion of character c^7 at position `p`. Initially, `c` was inserted at position `ip`. Transformations may change `p` but leave `ip` untouched.
- `Del(p)` represents the deletion of the character at position `p`.
- `Id()` acts as a no-op (to which operations may be transformed).

Object `Imine` extends the `ListOT` trait and implements the transformation functions (`Tii`, `Tid`, `Tdi`, `Tdd`) required for collaborative text editing (cf. Section 5.2). The implementation of these transformation functions is a straightforward translation from their description by Imine et al. [34]. The resulting object inherits automated proofs for TP_1 and TP_2 . When running these proofs, VeriFx reports that the transformation functions guarantee TP_1 but not TP_2 .

Based on the results shown in Table 2, we conclude that VeriFx is suited to verify other RDT families such as OT. Due to the number of cases that have to be considered, the verification times are longer than for CRDTs but are still acceptable for static verification [21].

7 Discussion

We now discuss the main design decisions behind VeriFx, its limitations and trade-offs.

Traits. For simplicity, VeriFx only supports single inheritance from traits. However, it could be extended to support multiple inheritance. Traits are not meant for subtyping because subtyping complicates verification as every subtype needs to be verified but these might not be known at compile time. Hence, class fields, method parameters, etc. cannot be of a trait type. Programmers can, however, define enumerations as these have a fixed number of constructors, which are known at compile time. Note that traits can define type parameters with upper type bounds. The type checker uses these bounds to ensure that every extending class or trait is well-typed. The compiled SMT program does not contain traits as they are compiled away (cf. Section 4.2). Proofs, classes, and methods cannot have bounds on type parameters because the compiler does not know all subtypes.

Functional collections. VeriFx encodes higher-order operations on collections (e.g. `map`, `filter`) using arrays, which are treated as function spaces in the Combinatory Array Logic (CAL) [23]. Hence, anonymous functions (lambdas) merely define arrays that are first-class. SMT solvers can efficiently reason about VeriFx’s functional collections because CAL is decidable. However, some operations are encoded using universal or existential quantifiers which may hamper decidability. In practice, VeriFx can verify RDTs involving complex functional operations. Unfortunately, VeriFx’s collections do not yet provide aggregation methods (e.g. `fold` and `reduce`) because this is beyond the capabilities of CAL. These restrictions may soon be lifted as SMT-LIB v3 [35] preliminary plans incorporate new theories that include aggregation functions such as `fold`.

Trade-off between expressiveness and verifiability. All constructs in VeriFx were carefully designed to have efficient SMT encoding. Overall, general loop constructs cannot be verified automatically as those require inductive proofs. A key insight of VeriFx is that *implicit* loop constructs (e.g. `map`, etc.) enable the automatic verification of an extensive portfolio of RDTs. Even though the language does not provide general loop constructs, programmers can define recursive methods. While VeriFx will not prove facts about (unbounded) recursive methods out-of-the-box, programmers can still verify these implementations by explicitly defining inductive proofs. This requires devising a suitable induction hypothesis and defining two proofs: one for the base case, and another for the induction step. Then, VeriFx can verify both proofs. This approach has been used to verify nested CRDT designs [13].

⁷ We represent characters using integers that correspond to their ASCII code.

8 Related Work

We focus our comparison of related work on verification languages and approaches for verifying RDTs, invariants in distributed systems, and operational transformation.

Verification languages. Verification languages can be classified into three categories: interactive, auto-active, and automated [45]. Interactive languages include proof assistants like Coq and Isabelle/HOL in which programmers define theorems and prove them manually using proof tactics. Although automation tactics exist, proving complex theorems requires considerable manual proof efforts. Vazou et al. [76] introduce the idea of refinement reflection in Liquid Haskell [75], where user-defined functions are reflected in a decidable fragment of SMT logic and can be used in refinement types to express correctness properties. Similarly, in VeriFx *every* construct of the language and its collections are reflected in SMT logic such that arbitrary VeriFx programs can be reflected in the logic. However, VeriFx enables *automated* verification of user-defined correctness properties whereas, Liquid Haskell requires programmers to express correctness properties using refinement types and *manually* write proofs as Haskell functions. Moreover, VeriFx offers an iterative process where incorrect designs are improved based on the counterexamples, whereas Liquid Haskell only raises a type error. Auto-active verification languages like Dafny [44] and Spec# [11] verify programs based on annotations provided by the programmer (e.g. preconditions, postconditions, loop invariants). Intermediate verification languages (IVLs) like Boogie [10] and Why3 [26] automate the proof task by generating verification conditions (VCs) from source code and discharging them using one or more SMT solvers. IVLs are not meant to be used by programmers directly. Instead, programs written in some verification language (e.g. Dafny, Spec#) are translated to an IVL to verify the VCs. Regarding automated verification, the work by Kaki and Jagannathan [37] integrated an automated verification framework in a refinement type system. Programmers write relational specifications that define structural relations for the RDT at hand and express correctness properties as refinement types atop operations. However, writing relational specifications for advanced data types is non-trivial and can be rather verbose, as noted by the authors themselves. In contrast, VeriFx does not require separate specifications.

Verifying SEC for RDTs. Burckhardt et al. [20] propose a formal framework that enables the specification and verification of RDTs. Attiya et al. [5] use a variation on this framework to provide specifications of replicated lists and prove the correctness of an existing text editing protocol. Gomes et al. [27] and Zeller et al. [79] propose formal frameworks in the Isabelle/HOL theorem prover to mechanically verify SEC for CRDTs. In contrast to VeriFx, Gomes et al. [27] only consider operation-based CRDTs but model the underlying network to reason about causal delivery of messages. Nieto et al. [59] developed libraries to implement and verify op-based CRDTs in separation logic. Their approach requires programmers to write Coq specifications atop the provided libraries and manually prove correctness. Liu et al. [54] extend Liquid Haskell with typeclass refinements and use them to prove SEC for some of their own CRDTs. While simple proofs can be discharged automatically by the underlying SMT solver, advanced CRDTs also require significant proof efforts (as discussed in Section 2). All the aforementioned verification techniques require significant effort and expertise whereas, VeriFx fully automatically verified 51 well-known CRDTs. Liang and Feng [51] propose a new correctness criterion for CRDTs that extends SEC with functional correctness and enables manual verification of CRDT implementations and client programs using them. They mainly focus on functional correctness and provide paper proofs rather than automated verification. Wang et al. [77]

propose replication-aware linearizability, a criterion that enables sequential reasoning to prove the correctness of CRDT implementations. The CRDTs were manually encoded in the Boogie verification tool to prove correctness. Those encodings are non-trivial and differ from real-world CRDT implementations. Nagar and Jagannathan [56] developed a proof rule that is parametrized by the consistency model and automatically checks convergence for CRDTs. Unfortunately, their framework introduces imprecision and may reject correct CRDTs. Moreover, their framework requires a first-order logic specification of the CRDT. In contrast, VeriFx can verify high-level CRDT implementations instead of specifications. Finally, Jagadeesan and Riely [36] introduce a notion of validity for RDTs and manually prove it for some CRDTs. We do not consider validity in this work.

Verifying applications invariants. Some work has focused on verifying application invariants under weak consistency. Bailis et al. [6] introduce invariant confluent operations that maintain application invariants, even without coordination. Whittaker and Hellerstein [78] devise a decision procedure for invariant confluence that can be checked automatically. Other work has focused on verifying invariants for RDTs [7, 29, 57, 58, 80]. Soteria [57] verifies program invariants for state-based RDTs. Repliss [80] verifies program invariants for applications that are built on top of their CRDT library. CISE [29, 58] proposes a proof rule to check that a chosen consistency level for operations preserves the application invariants. IPA [7] detects invariant-breaking operations and proposes changes to the operations in order to preserve the invariants. All these approaches assume that the underlying RDT is correct, while VeriFx enables programmers to verify that this is the case. This paper does not consider RDTs with mixed consistency levels as [24, 31, 46–48, 50, 55, 81, 82].

Verifying operational transformation functions. Ellis and Gibbs [25] first proposed an algorithm for OT together with a set of transformation functions. Several works [70, 72] showed that integration algorithms like adOPTed [64], SOCT2 [70], and GOTO [71] guarantee convergence iff the transformation functions satisfy the TP_1 and TP_2 properties. Ellis and Gibbs [25]’s functions do not satisfy these properties [64, 70, 72] and, over the years, several functions were proposed [64, 69, 72]. Imine et al. [34] used SPIKE, an automated theorem prover, to verify the correctness of these functions and found counterexamples for all of them, except for Suleiman et al. [69]’s functions. As shown in Section 6.2, we reproduced their study and generated similar counterexamples. Imine et al. [34] proposed a simpler set of functions which later was found to also violate TP_2 [49, 62]. VeriFx also found this counterexample.

9 Conclusion

To support the development of correct RDTs, we propose VeriFx, a high-level programming language powerful enough to implement RDTs as CRDTs and OT, and verify them automatically without requiring annotations or programmer intervention. Our approach enables programmers to implement RDTs, and express and verify correctness properties, all within the *same* language. This avoids gaps between the implementation and verification. VeriFx high-level counterexamples enable programmers to iteratively improve their implementation. Once verified, RDTs can be transpiled to mainstream languages, e.g. Scala and JavaScript.

VeriFx shows that automated verification based on SMT solving can verify real-world RDT implementations with minimal programmer intervention. This work accounts for the most extensive portfolio of mechanically verified RDTs to date, including 51 CRDTs and 9 OT designs. All were verified in a matter of seconds or minutes and with minimal effort.

In this work, we focused on verifying correctness properties in the domain of RDTs. In future work, we would like to explore the applicability of VeriF_x to other domains.

References

- 1 Deepthi Devaki Akkoorath and Annette Bieniusa. Antidote: The highly-available geo-replicated database with strongest guarantees. Technical report, Technical Report. Tech. U. Kaiserslautern., 2016.
- 2 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by delta-mutation. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Int. Conference on Networked Systems*, pages 62–76, Agadir, Morocco, 2015. Springer-Verlag.
- 3 AntidoteDB. Implementation of a Disable-Wins Flag CRDT in AntidoteDB. https://github.com/AntidoteDB/antidote/blob/master/apps/antidote_crdt/src/antidote_crdt_flag_dw.erl. Accessed: 2022-07-19.
- 4 AntidoteDB. Implementation of an Enable-Wins Flag CRDT in AntidoteDB. https://github.com/AntidoteDB/antidote/blob/master/apps/antidote_crdt/src/antidote_crdt_flag_ew.erl. Accessed: 2022-07-19.
- 5 Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 259–268, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933090.
- 6 Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014. doi:10.14778/2735508.2735509.
- 7 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. IPA: Invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, December 2018.
- 8 Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017. arXiv:1710.04469.
- 9 Carlos Baquero, Omer Katz, Brian Cannard, and Georges Younes. JGraphT: a Java library of graph theory data structures and algorithms. <https://github.com/CBaquero/delta-enabled-crdts>. Accessed: 22-11-2022.
- 10 Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 11 Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 12 Basho Technologies. Riak KV. <https://riak.com/products/riak-kv/index.html>. Accessed: 22-11-2022.
- 13 Jim Bauwens and Elisa Gonzalez Boix. Nested pure operation-based CRDTs. In *To Appear in 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, WA, LIPIcs*, 2023.
- 14 Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012. arXiv:1210.3368.
- 15 Sascha Böhme, Anthony C. J. Fox, Thomas Sewell, and Tjark Weber. Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 183–198, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

- 16 Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In *International Conference on Interactive Theorem Proving*, pages 179–194. Springer, 2010.
- 17 Eric Brewer. CAP twelve years later: How the “rules” have changed. *Computer*, 45:23–29, February 2012.
- 18 Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 7, New York, NY, USA, 2000. ACM. doi:10.1145/343477.343502.
- 19 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag.
- 20 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. doi:10.1145/2535838.2535848.
- 21 Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.
- 22 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 23 Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *2009 Formal Methods in Computer-Aided Design*, pages 45–52, 2009. doi:10.1109/FMCD.2009.5351142.
- 24 Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. ECROs: Building Global Scale Systems from Sequential Code. *Proc. ACM Program. Lang.*, 5(OOPSLA), November 2021.
- 25 C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407, New York, NY, USA, 1989. ACM. doi:10.1145/67544.66963.
- 26 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- 27 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133933.
- 28 Google. Protocol buffers. <https://developers.google.com/protocol-buffers>. Accessed: 10-10-2022.
- 29 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. *SIGPLAN Not.*, 51(1):371–384, January 2016. doi:10.1145/2914770.2837625.
- 30 Pat Helland. Immutability changes everything: We need it, we can afford it, and the time is now. *Queue*, 13(9):101–125, November 2015. doi:10.1145/2857274.2884038.
- 31 Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290387.
- 32 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.
- 33 Abdessamad Imine. Exchange of mails regarding OT, and unpublished register and stack designs. personal communication.
- 34 Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the Eighth Conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 277–293, USA, 2003. Kluwer Academic Publishers.

- 35 The SMT-LIB Initiative. SMT-LIB Version 3.0 - Preliminary Proposal. <http://smtlib.cs.uiowa.edu/version3.shtml>. Accessed: 23-11-2022.
- 36 Radha Jagadeesan and James Riely. Eventual consistency for CRDTs. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 968–995, Cham, 2018. Springer International Publishing.
- 37 Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. *SIGPLAN Not.*, 49(9):311–324, August 2014. doi:10.1145/2692915.2628159.
- 38 Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360580.
- 39 Martin Kleppmann. A critique of the CAP theorem. *CoRR*, abs/1509.05393, 2015. arXiv:1509.05393.
- 40 Martin Kleppmann. Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode. Technical Report UCAM-CL-TR-969, University of Cambridge, Computer Laboratory, May 2022. doi:10.48456/tr-969.
- 41 Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Trans. on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.
- 42 Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: Synthesizing CRDTs with verified lifting. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563336.
- 43 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.
- 44 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 45 K. Rustan M. Leino and Michal Moskal. Usable auto-active verification. In *Usable Verification Workshop*, 2010.
- 46 Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC’14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.
- 47 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI’12, pages 265–278, USA, 2012. USENIX Association.
- 48 Cheng Li, Nuno Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 359–372, Boston, MA, 2018. USENIX Association.
- 49 Du Li and Rui Li. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, CSCW ’04, pages 457–466, New York, NY, USA, 2004. ACM. doi:10.1145/1031607.1031683.
- 50 Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, pages 324–349, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53288-8_16.
- 51 Hongjin Liang and Xinyu Feng. Abstraction for conflict-free replicated data types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 636–650, New York, NY, USA, 2021. ACM. doi:10.1145/3453483.3454067.
- 52 Lightbend, Inc. Akka. <https://akka.io/>. Accessed: 22-11-2022.
- 53 Lightbend Inc. Serialization. <https://doc.akka.io/docs/akka/current/serialization.html>. Accessed: 10-10-2022.

- 54 Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in liquid Haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428284.
- 55 Mae Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 226–241, New York, NY, USA, 2018. ACM. doi:10.1145/3192366.3192375.
- 56 Kartik Nagar and Suresh Jagannathan. Automated parameterized verification of CRDTs. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 459–477, Cham, 2019. Springer International Publishing.
- 57 Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In *Programming Languages and Systems: 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings*, pages 544–571, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-44914-8_20.
- 58 Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE tool: proving weakly-consistent applications correct. In Peter Alvaro and Alysso Bessani, editors, *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 2:1–2:3. ACM, 2016. doi:10.1145/2911151.2911160.
- 59 Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. Modular verification of op-based CRDTs in separation logic. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563351.
- 60 Peter W. O’Hearn. Continuous reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS ’18*, pages 13–25, New York, NY, USA, 2018. ACM. doi:10.1145/3209108.3209109.
- 61 OpenJDK. jmh - OpenJDK. <https://openjdk.java.net/projects/code-tools/jmh/>. Accessed: 13-05-2020.
- 62 Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 1–10, 2006. doi:10.1109/COLCOM.2006.361867.
- 63 Aurel Randolph, Hanifa Boucheneb, Abdessamad Imine, and Alejandro Quintero. On synthesizing a consistent operational transformation approach. *IEEE Transactions on Computers*, 64(4):1074–1089, 2015. doi:10.1109/TC.2014.2308203.
- 64 Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work, CSCW ’96*, pages 288–297, New York, NY, USA, 1996. ACM. doi:10.1145/240080.240305.
- 65 Scalameta. Scalameta: Library to read, analyze, transform and generate Scala programs. <https://scalameta.org/>. Accessed: 24-11-2022.
- 66 Marc Shapiro. Replicated Data Types. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia Of Database Systems*, volume Replicated Data Types, pages 1–5. Springer-Verlag, July 2017. doi:10.1007/978-1-4899-7993-3_80813-1.
- 67 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506, INRIA – Centre Paris-Rocquencourt, January 2011.
- 68 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems, SSS’11*, pages 386–400, Grenoble, France, 2011. Springer-Verslag.

- 69 Maher Suleiman, Michèle Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*, GROUP '97, pages 435–445, New York, NY, USA, 1997. ACM. doi:10.1145/266838.267369.
- 70 Maher Suleiman, Michèle Cart, and Jean Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 36–45, USA, 1998. IEEE Computer Society.
- 71 Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proc. of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68, 1998.
- 72 Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998. doi:10.1145/274444.274447.
- 73 The Apache Software Foundation. Cassandra: Open source NoSQL database. https://cassandra.apache.org/_/index.html. Accessed: 24-11-2022.
- 74 The SMT-LIB Initiative. SMT-LIB. <https://smtlib.cs.uiowa.edu>. Accessed: 24-11-2022.
- 75 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM. doi:10.1145/2628136.2628161.
- 76 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158141.
- 77 Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 980–993, New York, NY, USA, 2019. ACM. doi:10.1145/3314221.3314617.
- 78 Michael J. Whittaker and Joseph M. Hellerstein. Interactive checks for coordination avoidance. *Proc. VLDB Endow.*, 12(1):14–27, 2018. doi:10.14778/3275536.3275538.
- 79 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of CRDTs. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 33–48, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- 80 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Combining state- and event-based semantics to verify highly available programs. In Farhad Arbab and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software*, pages 213–232, Cham, 2020. Springer International Publishing.
- 81 Xin Zhao and Philipp Haller. Observable atomic consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 23–32, 2018.
- 82 Xin Zhao and Philipp Haller. Replicated data types that unify eventual consistency and observable atomic consistency. *Journal of Logical and Algebraic Methods in Programming*, 114:100561, 2020.

A VeriFx's Type System

We now present VeriFx's type system. An environment Γ is a partial and finite mapping from variables to types. A type environment Δ is a finite set of type variables. VeriFx's type system consists of a judgment for type wellformedness $\Delta \vdash T$ ok which says that type T is well-formed in context Δ , and a judgment for typing $\Delta; \Gamma \vdash e : T$ which says that in context Δ and environment Γ , the expression e is of type T . We abbreviate $\Delta \vdash T_1$ ok, \dots , $\Delta \vdash T_n$ ok to $\Delta \vdash \bar{T}$ ok, and $\Delta; \Gamma \vdash e_1 : T_1, \dots, \Delta; \Gamma \vdash e_n : T_n$ to $\Delta; \Gamma \vdash \bar{e} : \bar{T}$.

Below we define well-formed types:

$$\begin{array}{c}
\frac{}{\Delta \vdash \text{string ok}} \text{ (WF-STRING)} \quad \frac{}{\Delta \vdash \text{bool ok}} \text{ (WF-BOOL)} \quad \frac{}{\Delta \vdash \text{int ok}} \text{ (WF-INT)} \\
\\
\frac{\Delta \vdash \overline{T} \text{ ok} \quad \text{class } C \langle \overline{X} \rangle (\dots) \{ \dots \} \quad \text{or class } C \langle \overline{X} \rangle (\dots) \text{ extends } I \langle \dots \rangle \{ \dots \}}{\Delta \vdash C \langle \overline{T} \rangle \text{ ok}} \text{ (WF-CLASS)} \\
\\
\frac{\Delta \vdash \overline{T} \text{ ok} \quad \overline{T} <: \overline{P} \quad \text{trait } I \langle \overline{X} <: \overline{P} \rangle \{ \dots \} \quad \text{or trait } I \langle \overline{X} <: \overline{P} \rangle \text{ extends } I \langle \dots \rangle \{ \dots \}}{\Delta \vdash I \langle \overline{T} \rangle \text{ ok}} \text{ (WF-TRAIT)} \\
\\
\frac{\Delta \vdash \overline{T} \text{ ok} \quad \text{enum } E \langle \overline{X} \rangle \{ \dots \}}{\Delta \vdash E \langle \overline{T} \rangle \text{ ok}} \text{ (WF-ENUM)} \quad \frac{X \in \Delta}{\Delta \vdash X \text{ ok}} \text{ (WF-TVAR)}
\end{array}$$

Primitive types are always well-formed. A type variable X is valid if it is in scope: $X \in \Delta$, i.e. the surrounding method or class defined the type parameter. Class types and enumeration types are valid if a corresponding class or enumeration definition exists and all type arguments are well-formed.

We now define a few auxiliary definitions which are needed for the typing rules. The *fields* function takes a class type and returns its fields and their types:

$$\frac{\text{class } C \langle \overline{X} \rangle (\overline{v} : \overline{T}) \{ \overline{M} \} \quad \text{or class } C \langle \overline{X} \rangle (\overline{v} : \overline{T}) \text{ extends } I \langle \overline{Q} \rangle \{ \overline{M} \}}{\text{fields}(C \langle \overline{P} \rangle) = [\overline{P}/\overline{X}] \overline{v} : \overline{T}} \text{ (F-CLASS)}$$

The *ftypes* function takes an enumeration type and the name of one of its constructors and returns the type of the fields of that constructor.

$$\frac{\text{enum } E \langle \overline{X} \rangle \{ K(\overline{v} : \overline{T}), \dots \}}{\text{ftypes}(E \langle \overline{P} \rangle, K) = [\overline{P}/\overline{X}] \overline{T}} \text{ (FT-ENUM)}$$

The *mtype* function takes the name of a method and the type of a class, and returns the actual type signature of the method. If the method is not found in the class (MT-CLASS-REC rule) it is looked up in the hierarchy of super traits by the MT-TRAIT rules. For polymorphic methods, the returned type signature is polymorphic:

$$\frac{\text{class } C \langle \overline{X} \rangle (\dots) \{ \overline{M} \} \quad \text{or class } C \langle \overline{X} \rangle (\dots) \text{ extends } I \langle \overline{Q} \rangle \{ \overline{M} \} \quad \text{def } m \langle \overline{Y} \rangle (\overline{x} : \overline{T}) : T = e \in \overline{M}}{\text{mtype}(m, C \langle \overline{P} \rangle) = [\overline{P}/\overline{X}] (\langle \overline{Y} \rangle \overline{T} \rightarrow T)} \text{ (MT-CLASS)}$$

$$\frac{\text{class } C \langle \overline{X} \rangle (\dots) \text{ extends } I \langle \overline{Q} \rangle \{ \overline{M} \} \quad \text{def } m \langle \overline{Y} \rangle (\overline{x} : \overline{T}) : T = e \notin \overline{M}}{\text{mtype}(m, C \langle \overline{P} \rangle) = \text{mtype}(m, I \langle \overline{Q} \rangle)} \text{ (MT-CLASS-REC)}$$

$$\frac{\text{trait } I \langle \overline{X} <: \overline{T}' \rangle \{ \overline{M} \} \quad \text{or trait } I \langle \overline{X} <: \overline{T}' \rangle \text{ extends } I' \langle \dots \rangle \{ \overline{M} \} \quad \text{def } m \langle \overline{Y} \rangle (\overline{x} : \overline{T}) : T = e \in \overline{M}}{\text{mtype}(m, I \langle \overline{P} \rangle) = [\overline{P}/\overline{X}] (\langle \overline{Y} \rangle \overline{T} \rightarrow T)} \text{ (MT-TRAIT)}$$

$$\frac{\text{trait } I \langle \overline{X} <: \overline{T}' \rangle \{ \overline{M} \} \quad \text{or trait } I \langle \overline{X} <: \overline{T}' \rangle \text{ extends } I' \langle \overline{P} \rangle \{ \overline{M} \} \quad \text{def } m \langle \overline{Y} \rangle (\overline{x} : \overline{T}) : T = e \notin \overline{M}}{\text{mtype}(m, I \langle \overline{P} \rangle) = \text{mtype}(m, I' \langle \overline{P} \rangle)} \text{ (MT-TRAIT-REC)}$$

Similarly, we assume that there are functions $valNames(I(\overline{P}))$ and $declaredMethods(I(\overline{P}))$ that return all fields, respectively all methods, declared by a trait (and its super traits). The $ctors$ function takes an enumeration type and returns the names of its constructors.

$$\frac{\text{enum } E \langle \overline{X} \rangle \{ K(\overline{x} : \overline{T}) \}}{ctors(E \langle \overline{P} \rangle) = \overline{K}} \text{ (C-ENUM)}$$

Figure 6 shows the typing rules for expressions. Most rules are a simplification of Featherweight Generic Java [32] without subtyping. Quantified formulas are boolean expressions if their body also types to a boolean expression in the environment that is extended with the quantified variables (T-UNI and T-EXI rules). The logical implication is a well-typed boolean expression if both the antecedent and the consequent are boolean expressions (T-IMPL rule).

Classes are well-formed if the types of the fields are well-formed and all its methods are well-formed (T-CLASS1 rule). If the class extends a trait, it must also implement all fields and methods declared by the hierarchy of super traits (T-CLASS2 rule). The typing rules for trait definitions and object definitions can be defined similarly.

When instantiating an enumeration through one of its constructors $\text{new } K \langle \overline{P} \rangle (\overline{e})$, the provided arguments \overline{e} need to match the types of the constructors' fields, and the result effectively is an object of the enumeration type $E \langle \overline{P} \rangle$.

Programmers can pattern match on enumerations but the cases must be exhaustive, i.e. every constructor must be matched by at least one case. If all cases are of type T , then the resulting pattern match expression is also of type T .

Finally, the body of a proof must be a well-typed boolean expression.

B Core SMT Expressions

We will now discuss the expressions that are supported by Core SMT. Those expressions are common to most SMT solvers, except lambdas which, as mentioned before, are described by the preliminary proposal for SMT-LIB v3.0 and are only implemented by some SMT solvers such as Z3 [22].

Figure 7 provides an overview of all Core SMT expressions. The simplest expressions are literal values representing integers, strings, and booleans. Core SMT supports the typical arithmetic operators ($+$, $-$, $*$, $/$) and boolean operators (\wedge , \vee , and negation \neg) as well as universal and existential quantification, and logical implication. Immutable variables are defined by let bindings. Pattern matching is supported but the cases must be exhaustive. For example, when pattern matching against an algebraic data type every constructor must be handled. Core SMT supports two types of patterns: constructor patterns $n(\overline{n})$ that match a specific ADT constructor n and binds names to its fields \overline{n} , and wildcard patterns that match anything and give it a name n . References v refer to variables that are in scope, e.g. function parameters or variables introduced by let binding or pattern matching. If statements are supported but an else branch is mandatory and both branches must evaluate to the same sort. Functions can be called and type arguments can be provided explicitly to disambiguate polymorphic functions. For example, we defined an ADT $\text{Option} \langle T \rangle$ with two constructors `Some` and `None`. When calling the `None` constructor we need to explicitly provide a type argument since it cannot be inferred from the call, e.g. `None(int)()`. Finally, fields of an ADT can be accessed by their name. Arrays and lambdas were already discussed in Section 4.1.

$$\begin{array}{c}
\frac{}{\Delta; \Gamma \vdash \text{num} : \text{int}} \text{(T-NUM)} \quad \frac{}{\Delta; \Gamma \vdash \text{str} : \text{string}} \text{(T-STR)} \quad \frac{}{\Delta; \Gamma \vdash \text{true} : \text{bool}} \text{(T-TRUE)} \\
\frac{}{\Delta; \Gamma \vdash \text{false} : \text{bool}} \text{(T-FALSE)} \quad \frac{x \in \text{dom}(\Gamma)}{\Delta; \Gamma \vdash x : \Gamma(x)} \text{(T-VAR)} \quad \frac{\Delta; \Gamma \vdash e : \text{bool}}{\Delta; \Gamma \vdash !e : \text{bool}} \text{(T-NEG)} \\
\frac{\Delta; \Gamma \vdash e_1 : \text{int} \quad \Delta; \Gamma \vdash e_2 : \text{int}}{\Delta; \Gamma \vdash e_1 \oplus e_2 : \text{int}} \text{(T-OP1)} \quad \frac{\Delta; \Gamma \vdash e_1 : \text{bool} \quad \Delta; \Gamma \vdash e_2 : \text{bool}}{\Delta; \Gamma \vdash e_1 \otimes e_2 : \text{bool}} \text{(T-OP2)} \\
\frac{\Delta; \Gamma \vdash e_1 : \text{bool} \quad \Delta; \Gamma \vdash e_2 : T}{\Delta; \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(T-IF)} \quad \frac{\Delta \vdash T_1 \text{ ok} \quad \Delta; \Gamma \vdash e_1 : T_1 \quad \Delta; \Gamma, x : T_1 \vdash e_2 : T_2}{\Delta; \Gamma \vdash \text{val } x : T_1 = e_1 \text{ in } e_2 : T_2} \text{(T-VAL)} \\
\frac{\Delta \vdash \bar{T} \text{ ok} \quad \Delta; \Gamma, \bar{x} : \bar{T} \vdash e : T}{\Delta; \Gamma \vdash (\bar{x} : \bar{T}) \Rightarrow e : \bar{T} \rightarrow T} \text{(T-ABS)} \quad \frac{\Delta; \Gamma \vdash e_1 : \bar{T} \rightarrow T \quad \Delta; \Gamma \vdash \bar{e}_2 : \bar{T}}{\Delta; \Gamma \vdash e_1(\bar{e}_2) : T} \text{(T-CALL)} \\
\frac{\text{fields}(C(\bar{P})) = \bar{v} : \bar{T} \quad \Delta \vdash C(\bar{P}) \text{ ok} \quad \Delta; \Gamma \vdash \bar{e} : \bar{T}}{\Delta; \Gamma \vdash \text{new } C(\bar{P})(\bar{e}) : C(\bar{P})} \text{(T-N-CLASS)} \quad \frac{\Delta; \Gamma \vdash e : T_o \quad \text{fields}(T_o) = \bar{v} : \bar{T}}{\Delta; \Gamma \vdash e.v_i : T_i} \text{(T-FIELD)} \\
\frac{\Delta; \Gamma \vdash e_o : T_o \quad \Delta \vdash \bar{P} \text{ ok} \quad \text{mtype}(m, T_o) = \langle \bar{X} \rangle \bar{T} \rightarrow T \quad \Delta; \Gamma \vdash \bar{e} : [\bar{P}/\bar{X}] \bar{T}}{\Delta; \Gamma \vdash e_o.m \langle \bar{P} \rangle (\bar{e}) : [\bar{P}/\bar{X}] T} \text{(T-INV)} \quad \frac{\Delta \vdash \bar{T} \text{ ok} \quad \Delta; \Gamma, \bar{x} : \bar{T} \vdash e : \text{bool}}{\Delta; \Gamma \vdash \text{forall } (\bar{x} : \bar{T}), e : \text{bool}} \text{(T-UNI)} \\
\frac{\Delta \vdash \bar{T} \text{ ok} \quad \Delta; \Gamma, \bar{x} : \bar{T} \vdash e : \text{bool}}{\Delta; \Gamma \vdash \text{exists } (\bar{x} : \bar{T}), e : \text{bool}} \text{(T-EXI)} \quad \frac{\Delta; \Gamma \vdash e_1 : \text{bool} \quad \Delta; \Gamma \vdash e_2 : \text{bool}}{\Delta; \Gamma \vdash e_1 \Rightarrow e_2 : \text{bool}} \text{(T-IMPL)} \\
\frac{\text{ctors}(E(\bar{P})) = \bar{K} \quad K \in \bar{K} \quad \text{ftypes}(E(\bar{P}), K) = \bar{T} \quad \Delta \vdash E(\bar{P}) \text{ ok} \quad \Delta; \Gamma \vdash \bar{e} : \bar{T}}{\Delta; \Gamma \vdash \text{new } K(\bar{P})(\bar{e}) : E(\bar{P})} \text{(T-N-ENUM)} \\
\frac{\Delta; \Gamma \vdash e_0 : E(\bar{P}) \quad (\text{ctors}(E(\bar{P})) \setminus \hat{c} = \emptyset) \vee (\text{case } x \Rightarrow e \in \hat{c}) \vee (\text{case } _ \Rightarrow e \in \hat{c}) \text{ for each } c \in \hat{c} : \Delta; \Gamma \vdash c : T \text{ IN } e_0 \text{ match } \{ \dots \}}{\Delta; \Gamma \vdash e_0 \text{ match } \{ \hat{c} \} : T} \text{(T-MATCH)} \\
\frac{\Delta; \Gamma \vdash e_0 : E(\bar{P}) \quad \text{ftypes}(E(\bar{P}), K) = \bar{Q} \quad \Delta; \Gamma, \bar{x} : \bar{Q} \vdash e : T}{\Delta; \Gamma \vdash \text{case } K(\bar{x}) \Rightarrow e : T \text{ IN } e_0 \text{ match } \{ \dots \}} \text{(T-CTOR-PTN)} \\
\frac{\Delta; \Gamma \vdash e_0 : E(\bar{P}) \quad \Delta; \Gamma, x : E(\bar{P}) \vdash e : T}{\Delta; \Gamma \vdash \text{case } x \Rightarrow e : T \text{ IN } e_0 \text{ match } \{ \dots \}} \text{(T-NAMED-PTN)} \\
\frac{\Delta; \Gamma \vdash e : T}{\Delta; \Gamma \vdash \text{case } _ \Rightarrow e : T \text{ IN } e_0 \text{ match } \{ \dots \}} \text{(T-WCARD-PTN)} \\
\frac{\Delta = \bar{X} \quad \Delta \vdash \bar{T} \text{ ok}}{\text{enum } E(\bar{X}) \{ K(\bar{v} : \bar{T}) \} \text{ OK}} \text{(T-ENUM)} \\
\frac{\Delta = \bar{X}, \bar{Y} \quad \Delta \vdash \bar{T}, T \text{ ok} \quad \text{class } C(\bar{X})(\dots) \{ \dots \} \text{ or } \text{trait } C(\bar{X} <: \bar{Q}) \{ \dots \} \text{ or } \text{trait } C(\bar{X} <: \bar{Q}) \text{ extends } \dots \{ \dots \}}{\Delta; \bar{x} : \bar{T}, \text{this} : C(\bar{X}) \vdash e : T} \text{(T-METHOD)} \\
\frac{\Delta = \bar{X} \quad \Delta; \emptyset \vdash e : \text{bool}}{\text{proof } p(\bar{X}) \{ e \} \text{ OK}} \text{(T-PROOF)} \quad \frac{\Delta = \bar{X} \quad \Delta \vdash \bar{T} \text{ ok} \quad \bar{M} \text{ OK IN } C(\bar{X})}{\text{class } C(\bar{X})(\bar{v} : \bar{T}) \{ \bar{M} \} \text{ OK}} \text{(T-CLASS1)} \\
\frac{\Delta = \bar{X} \quad \Delta \vdash \bar{T} \text{ ok} \quad \Delta \vdash I(\bar{P}) \text{ ok} \quad \text{trait } I(\dots) \{ B \} \text{ or } \text{trait } I(\dots) \text{ extends } \dots \{ B \} \quad \text{valNames}(I(\bar{P})) \subset \bar{v} \quad \text{declaredMethods}(I(\bar{P})) \subset \bar{M} \quad \bar{M} \text{ OK IN } C(\bar{X})}{\text{class } C(\bar{X})(\bar{v} : \bar{T}) \text{ extends } I(\bar{P}) \{ \bar{M} \} \text{ OK}} \text{(T-CLASS2)} \\
\frac{\Delta = \bar{X} \quad \Delta \vdash \bar{T} \text{ ok} \quad \Delta \vdash I'(\bar{P}) \text{ ok} \quad \text{trait } I'(\dots) \{ \dots \} \text{ or } \text{trait } I'(\dots) \text{ extends } \dots \{ \dots \} \quad B = \text{val}\bar{D} \cup \text{method}\bar{D} \cup \bar{M} \quad \bar{M} \text{ OK IN } I(\bar{X}) \quad \text{valNames}(I'(\bar{P})) \subset \text{val}\bar{D} \quad \text{declaredMethods}(I'(\bar{P})) \subset (\text{method}\bar{D} \cup \bar{M})}{\text{trait } I(\bar{X} <: \bar{T}) \text{ extends } I'(\bar{P}) \{ \bar{B} \} \text{ OK}} \text{(T-TRAIT)}
\end{array}$$

■ **Figure 6** Typing VeriF_x expressions.

$e ::= \text{num} \mid \text{str} \mid \text{true} \mid \text{false}$	<i>(primitive values)</i>
$e[\tilde{e}] \mid e[\tilde{e}] := e \mid \lambda(\tilde{x} : \tilde{T}).e$	
$x \mid e \oplus e \mid e \otimes e \mid \neg e$	
$\text{match}(e, \text{case}(\text{ptn}, e))$	<i>(pattern matching)</i>
$\text{let } x = e \text{ in } e$	<i>(let expression)</i>
$\text{if}(e, e, e)$	<i>(conditional expression)</i>
$e(e)$	<i>(function call)</i>
$f\langle\tilde{T}\rangle(e)$	<i>(function call with explicit type arguments)</i>
$e.v$	<i>(field access)</i>
$\forall(\tilde{x} : \tilde{T}).e \mid \exists(\tilde{x} : \tilde{T}).e$	<i>(quantified formulas)</i>
$e \implies e$	<i>(logical implication)</i>
$\text{ptn} ::= K(\tilde{x}) \mid x$	<i>(patterns)</i>

■ **Figure 7** All Core SMT expressions.

C Compiler Semantics

We now discuss the compiler semantics that was not discussed in the main body of the paper. First, we provide all compilation rules for expressions in Appendix C.1. Then, we provide all compilation rules for sets and maps in Appendices C.2 and C.3 respectively.

C.1 Compiling Expressions

Figure 8 shows the compilation rules for expressions. The operands of binary operators \oplus are compiled recursively. A negated expression is compiled to the negation of the compiled expression. For if statements, the condition, and both branches are compiled recursively. In VeriF_x, **this** can be used inside the body of a method to refer to the current object. The reference is compiled to a similar *this* reference in Core SMT which refers to the *this* parameter which is always the first parameter of any method (cf. compilation of class methods in Section 4.2). We explained how to compile the remaining expressions in Section 4.2.

Figure 9 shows the compilation rules for logic expressions which in VeriF_x can only occur within the body of proofs. For quantified formulas the types of the variables \tilde{T} and the formula e are compiled. For logical implications, the antecedent and the consequent are compiled recursively.

Finally, pattern match expressions are compiled to similar pattern match expressions in Core SMT (shown in Figure 10). To this end, every pattern is compiled recursively. Core SMT supports two types of patterns: constructor patterns $n_1(\overline{n_2})$ that match an algebraic data type constructor n_1 and bind its fields to the provided names $\overline{n_2}$, and wildcard patterns n that match any value and give it a name n . Every VeriF_x pattern is compiled into the corresponding Core SMT pattern. The first pattern, $n_1(\overline{n_2})$, matches an ADT constructor n_1 and binds its fields to $\overline{n_2}$. It is compiled to an equivalent constructor pattern in Core SMT. The other two patterns match any expression and are compiled to an equivalent wildcard pattern in Core SMT.

C.2 Compiling Sets

In Section 4.4 we explained how basic set operations (**add**, **remove**, **contains**) and some advanced operations (**filter**, **map**) are compiled to Core SMT. Now, we explain how the remaining operations on sets are compiled. Figure 11 shows the compilation rules for

$\llbracket x \rrbracket$	$= x$
$\llbracket e_1 \oplus e_2 \rrbracket$	$= \llbracket e_1 \rrbracket \oplus \llbracket e_2 \rrbracket$
$\llbracket !e \rrbracket$	$= \neg \llbracket e \rrbracket$
$\llbracket \text{val } x : T = e_1 \text{ in } e_2 \rrbracket$	$= \text{let } x = \llbracket e_1 \rrbracket \text{ in } \llbracket e_2 \rrbracket$
$\llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket$	$= \text{if } (\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket, \llbracket e_3 \rrbracket)$
$\llbracket (\bar{x} : \bar{T}) \Rightarrow e \rrbracket$	$= \lambda(\bar{x} : \llbracket \bar{T} \rrbracket_t) . \llbracket e \rrbracket$
$\llbracket e_1(\bar{e}_2) \rrbracket$	$= \llbracket e_1 \rrbracket[\llbracket \bar{e}_2 \rrbracket]$
$\llbracket \text{new Set } \langle T \rangle () \rrbracket$	$= \lambda(x : \llbracket T \rrbracket_t) . \text{false}$
$\llbracket \text{new Map } \langle T, P \rangle () \rrbracket$	$= \lambda(x : \llbracket T \rrbracket_t) . \text{None}(\llbracket P \rrbracket_t)()$
$\llbracket \text{new } C \langle \bar{T} \rangle (\bar{e}) \rrbracket$	$= C'(\llbracket \bar{T} \rrbracket_t)(\llbracket \bar{e} \rrbracket)$
	where $C' = \text{str_concat}(C, \text{"_ctor"})$
$\llbracket \text{new } K \langle \bar{T} \rangle (\bar{e}) \rrbracket$	$= K(\llbracket \bar{T} \rrbracket_t)(\llbracket \bar{e} \rrbracket)$
$\llbracket e.v \rrbracket$	$= \llbracket e \rrbracket.v$
$\llbracket e_1.m \langle \bar{T} \rangle (\bar{e}) \rrbracket$	$= m'(\llbracket \bar{P} \rrbracket_t, \llbracket \bar{T} \rrbracket_t)(\llbracket e_1 \rrbracket, \llbracket \bar{e} \rrbracket)$
	where $\text{typeof}(e_1) = C(\bar{P})$
	and $m' = \text{str_concat}(C, \text{"_"}, m)$ and $\bar{P} \cap \bar{T} = \emptyset$

■ **Figure 8** Compiling expressions.

$\llbracket \text{forall } (\bar{x} : \bar{T}) . e \rrbracket$	$= \forall(\bar{x} : \llbracket \bar{T} \rrbracket_t) . \llbracket e \rrbracket$
$\llbracket \text{exists } (\bar{x} : \bar{T}) . e \rrbracket$	$= \exists(\bar{x} : \llbracket \bar{T} \rrbracket_t) . \llbracket e \rrbracket$
$\llbracket e_1 \Longrightarrow e_2 \rrbracket$	$= \llbracket e_1 \rrbracket \Longrightarrow \llbracket e_2 \rrbracket$

■ **Figure 9** Compiling logical expressions.

operations over sets. The union of two sets e_1 and e_2 is compiled to a lambda which defines an array of elements x of type $\llbracket T \rrbracket_t$ containing only elements that are in at least one of the two sets, i.e. $\llbracket e_1 \rrbracket[x] \vee \llbracket e_2 \rrbracket[x]$. Similarly, the intersection of two sets e_1 and e_2 is compiled to a lambda which defines an array containing only elements that are in both sets, i.e. $\llbracket e_1 \rrbracket[x] \wedge \llbracket e_2 \rrbracket[x]$. For set difference, the lambda defines an array containing only elements that are in e_1 and not in e_2 . A set e_1 is a subset of e_2 iff all elements from e_1 are also in e_2 . A set e is non-empty if an element x exists that is in the set, i.e. $\llbracket e \rrbracket[x]$. A set e is empty if all elements x are not in the set. A predicate $e_2 : T \rightarrow \text{bool}$ holds for all elements of a set e_1 if for every element x that is in the set the predicate is true, i.e. $\llbracket e_1 \rrbracket[x] \Longrightarrow \llbracket e_2 \rrbracket[x]$. A predicate $e_2 : T \rightarrow \text{bool}$ holds for at least one element of a set e_1 if there exists an element x that is in the set and for which the predicate holds, i.e. $\llbracket e_1 \rrbracket[x] \wedge \llbracket e_2 \rrbracket[x]$.

C.3 Compiling Maps

Maps are encoded as arrays from the key type to an optional value:

$$\llbracket \text{Map } \langle T, P \rangle \rrbracket_t = \text{Array}(\llbracket T \rrbracket_t, \text{Option}(\llbracket P \rrbracket_t))$$

Optional values indicate the presence or absence of a value for a certain key. The option type is defined as an ADT with two constructors: `Some(value)` which holds a value and `None()` indicating the absence of a value. An empty map corresponds to an array containing `None()` for every key and is created by a lambda that returns `None()` for every key:

$$\begin{aligned}
 \llbracket e \text{ match } \{ \text{case } r \Rightarrow e_c \} \rrbracket &= \text{match}(\llbracket e \rrbracket, \text{pat}[\llbracket \text{case } r \Rightarrow e_c \rrbracket]) \\
 \text{pat}[\llbracket \text{case } K(\bar{x}) \Rightarrow e \rrbracket] &= \text{case}(K(\bar{x}), \llbracket e \rrbracket) \\
 \text{pat}[\llbracket \text{case } x \Rightarrow e \rrbracket] &= \text{case}(x, \llbracket e \rrbracket) \\
 \text{pat}[\llbracket \text{case } _ \Rightarrow e \rrbracket] &= \text{case}(_, \llbracket e \rrbracket)
 \end{aligned}$$

■ **Figure 10** Compiling pattern match expressions.

$$\begin{aligned}
 \llbracket e_1.\text{add}(e_2) \rrbracket &= \llbracket e_1 \rrbracket[\llbracket e_2 \rrbracket] := \text{true} \\
 \llbracket e_1.\text{remove}(e_2) \rrbracket &= \llbracket e_1 \rrbracket[\llbracket e_2 \rrbracket] := \text{false} \\
 \llbracket e_1.\text{contains}(e_2) \rrbracket &= \llbracket e_1 \rrbracket[\llbracket e_2 \rrbracket] \\
 \llbracket e_1.\text{filter}(e_2) \rrbracket &= \lambda(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \wedge \llbracket e_2 \rrbracket[x] \quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \\
 \llbracket e_1.\text{map}(e_2) \rrbracket &= \lambda(y : \llbracket P \rrbracket_t). \exists(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \wedge \llbracket e_2 \rrbracket[x] = y \\
 &\quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \wedge \text{typeof}(e_2) = T \rightarrow P \\
 \llbracket e_1.\text{union}(e_2) \rrbracket &= \lambda(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \vee \llbracket e_2 \rrbracket[x] \\
 &\quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \wedge \text{typeof}(e_2) = \text{Set}\langle T \rangle \\
 \llbracket e_1.\text{intersect}(e_2) \rrbracket &= \lambda(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \wedge \llbracket e_2 \rrbracket[x] \\
 &\quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \wedge \text{typeof}(e_2) = \text{Set}\langle T \rangle \\
 \llbracket e_1.\text{diff}(e_2) \rrbracket &= \lambda(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \wedge \neg \llbracket e_2 \rrbracket[x] \\
 &\quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \wedge \text{typeof}(e_2) = \text{Set}\langle T \rangle \\
 \llbracket e_1.\text{subsetOf}(e_2) \rrbracket &= \forall(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \implies \llbracket e_2 \rrbracket[x] \\
 &\quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \wedge \text{typeof}(e_2) = \text{Set}\langle T \rangle \\
 \llbracket e_1.\text{nonEmpty}() \rrbracket &= \exists(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \quad \text{where } \text{typeof}(e) = \text{Set}\langle T \rangle \\
 \llbracket e_1.\text{isEmpty}() \rrbracket &= \forall(x : \llbracket T \rrbracket_t). \neg \llbracket e_1 \rrbracket[x] \quad \text{where } \text{typeof}(e) = \text{Set}\langle T \rangle \\
 \llbracket e_1.\text{forall}(e_p) \rrbracket &= \forall(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \implies \llbracket e_p \rrbracket[x] \\
 &\quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \text{ and } \text{typeof}(e_p) = T \rightarrow \text{bool} \\
 \llbracket e_1.\text{exists}(e_p) \rrbracket &= \exists(x : \llbracket T \rrbracket_t). \llbracket e_1 \rrbracket[x] \wedge \llbracket e_p \rrbracket[x] \\
 &\quad \text{where } \text{typeof}(e_1) = \text{Set}\langle T \rangle \text{ and } \text{typeof}(e_p) = T \rightarrow \text{bool}
 \end{aligned}$$

■ **Figure 11** Compiling set operations.

$$\llbracket \text{new Map} \langle T, P \rangle () \rrbracket = \lambda(x : \llbracket T \rrbracket_t). \text{None}(\llbracket P \rrbracket_t)()$$

Operations on maps are compiled as follows:

$$\begin{aligned}
 \text{map}[\llbracket e_m.\text{add}(e_k, e_v) \rrbracket] &= \llbracket e_m \rrbracket[\llbracket e_k \rrbracket] := \text{Some}(\llbracket e_v \rrbracket) \\
 \text{map}[\llbracket e_m.\text{remove}(e_k) \rrbracket] &= \llbracket e_m \rrbracket[\llbracket e_k \rrbracket] := \text{None}(\llbracket V \rrbracket_t)() \\
 \text{map}[\llbracket e_m.\text{contains}(e_k) \rrbracket] &= \llbracket e_m \rrbracket[\llbracket e_k \rrbracket] \neq \text{None}(\llbracket V \rrbracket_t)() \\
 \text{map}[\llbracket e_m.\text{get}(e_k) \rrbracket] &= \llbracket e_m \rrbracket[\llbracket e_k \rrbracket].\text{value} \\
 \text{map}[\llbracket e_m.\text{getOrElse}(e_k, e_v) \rrbracket] &= \text{if}(\llbracket e_m \rrbracket[\llbracket e_k \rrbracket] = \text{None}(\llbracket V \rrbracket_t)(), \llbracket e_v \rrbracket, \llbracket e_m \rrbracket[\llbracket e_k \rrbracket].\text{value})
 \end{aligned}$$

A key-value pair $e_k \mapsto e_v$ is added to a map e_m by updating the entry for the compiled key $\llbracket e_k \rrbracket$ in the compiled array $\llbracket e_m \rrbracket$ with the compiled value, $\text{Some}(\llbracket e_v \rrbracket)$. A key e_k is removed from a map e_m by updating the corresponding entry to $\text{None}(\llbracket V \rrbracket_t)()$, thereby indicating the absence of a value. Note that None is polymorphic but the type parameter cannot be inferred from the arguments; it is thus passed explicitly. A key e_k is present in a map e_m if the value that is associated to the key is not $\text{None}(\llbracket V \rrbracket_t)()$. The `get` method fetches the value that is associated to a key e_k in a map e_m . To this end, the compiled key $\llbracket e_k \rrbracket$ is accessed in the compiled map $\llbracket e_m \rrbracket$ and the value it holds is then fetched by accessing the `value` field of

the `Some` constructor. Even though the entry that is read from the array is an option type (i.e. a `None` or a `Some`) we can access the `value` field because the interpretation of `value` is underspecified in SMT. If the entry is a `None`, the SMT solver can assign any interpretation to the `value` field. Hence, the `get` method on maps should only be called if the key is known to be present in the map, e.g. after calling `contains`. VeriF_x also features a safe variant, called `getOrElse`, which returns a default value if the key is not present.

Next, we explain how to encode the advanced map operations. Figure 12 defines the SMT encoding for all advanced map operations. The `keys` method on maps returns a set containing only the keys that are present in the map. Calls to `keys` on a map e_m of type $\text{Map} \langle K, V \rangle$ are compiled to a lambda which defines a set of keys k of the compiled key type $\llbracket K \rrbracket_t$ such that a key is present in the set iff it is present in the compiled map: $\llbracket e_m \rrbracket [k] \neq \text{None}(\llbracket V \rrbracket_t)()$. A predicate e_p of type $(K, V) \rightarrow \text{bool}$ holds for all elements of a map e_m of type $\text{Map} \langle K, V \rangle$ iff it holds for every key k that is present in the map and its associated value:

$$\underbrace{\llbracket e_m \rrbracket [k] \neq \text{None}(\llbracket V \rrbracket_t)()}_{e_m.\text{contains}(k)} \implies \underbrace{\llbracket e_p \rrbracket [k, \llbracket e_m \rrbracket [k].\text{value}]}_{e_p(k, e_m.\text{get}(k))}$$

Similarly, the `values` method returns a set with all values of the map. To this end, it defines an array containing all values for which at least one key exists that maps to that value.

A predicate e_p of type $(K, V) \rightarrow \text{bool}$ holds for at least one element of a map e_m of type $\text{Map} \langle K, V \rangle$ iff there exists a key k with associated value v that is present in the map and for which the predicate holds. Mapping a function e_f over the key-value pairs of a map e_m is encoded as a lambda that defines an array containing only the keys that are present in the compiled map $\llbracket e_m \rrbracket$ and whose values are the result of applying e_f on the original value, i.e. $\text{Some}(\llbracket e_f \rrbracket [k, \llbracket e_m \rrbracket [k].\text{value}])$. The `mapValues` method is similar except that it applies the provided function only on the value. A map e_m can be filtered using a predicate e_p such that the resulting map only contains key-value pairs that fulfill the predicate. Calls to `filter` are encoded as a lambda that defines an array containing only the key-value pairs that are in the compiled map:

$$\text{if} \left(\underbrace{\llbracket e_m \rrbracket [k] \neq \text{None}(\llbracket V \rrbracket_t)()}_{\text{in original map}} \wedge \underbrace{\llbracket e_p \rrbracket [k, \llbracket e_m \rrbracket [k].\text{value}]}_{\text{predicate holds}}, \right. \\ \left. \underbrace{\text{Some}(\llbracket e_m \rrbracket [k].\text{value})}_{\text{then keep the value}}, \underbrace{\text{None}(\llbracket V \rrbracket_t)()}_{\text{else not in the map}} \right)$$

To zip two maps e_{m_1} and e_{m_2} the compiler creates a lambda that defines an array containing only the keys that are present in both maps and the value is a tuple holding the corresponding values from both maps:

$$\text{Some}(\text{Tuple_ctor}(\llbracket e_{m_1} \rrbracket [k].\text{value}, \llbracket e_{m_2} \rrbracket [k].\text{value}))$$

To combine two maps e_{m_1} and e_{m_2} with a function e_f the compiler creates a lambda that defines an array containing all the keys from e_{m_1} and e_{m_2} . If a key is present in both maps their values are combined using the provided function e_f :

$$\text{Some}(\llbracket e_f \rrbracket [\llbracket e_{m_1} \rrbracket [k].\text{value}, \llbracket e_{m_2} \rrbracket [k].\text{value}])$$

If a key-value pair is present in only one of the maps it is also present in the new map. If a key is not present in e_{m_1} neither in e_{m_2} then it is also not present in the resulting map.

Vectors and Lists. The encoding of sets and maps is very useful to build new data structures in VeriF_x without having to encode them manually in SMT. For example, vectors and lists are implemented on top of maps. Internally, they map indices between 0 and $\text{size} - 1$

$$\begin{aligned}
\text{map}[e_m.\text{keys}()] &= \lambda(x : \llbracket K \rrbracket_t). \llbracket e_m \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)() \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \\
\text{map}[e_m.\text{values}()] &= \lambda(x : \llbracket V \rrbracket_t). \exists(k : \llbracket K \rrbracket_t). \llbracket e_m \rrbracket [k] = \text{Some}(x) \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \\
\text{map}[e_m.\text{bijective}()] &= \forall(k_1 : \llbracket K \rrbracket_t, k_2 : \llbracket K \rrbracket_t). \\
&\quad (k_1 \neq k_2 \wedge \llbracket e_m \rrbracket [k_1] \neq \text{None}(\llbracket V \rrbracket_t)() \wedge \llbracket e_m \rrbracket [k_2] \neq \text{None}(\llbracket V \rrbracket_t)()) \\
&\quad \implies \llbracket e_m \rrbracket [k_1] \neq \llbracket e_m \rrbracket [k_2] \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \\
\text{map}[e_m.\text{forall}(e_p)] &= \forall(x : \llbracket K \rrbracket_t). \llbracket e_m \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)() \implies \llbracket e_p \rrbracket [x, \llbracket e_m \rrbracket [x].\text{value}] \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_p) = (K, V) \rightarrow \text{bool} \\
\text{map}[e_m.\text{exists}(e_p)] &= \exists(x : \llbracket K \rrbracket_t). \llbracket e_m \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)() \wedge \llbracket e_p \rrbracket [x, \llbracket e_m \rrbracket [x].\text{value}] \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_p) = (K, V) \rightarrow \text{bool} \\
\text{map}[e_m.\text{map}(e_f)] &= \lambda(x : \llbracket K \rrbracket_t). \text{if}(\llbracket e_m \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)(), \\
&\quad \text{Some}(\llbracket e_f \rrbracket [x, \llbracket e_m \rrbracket [x].\text{value}]), \\
&\quad \text{None}(\llbracket W \rrbracket_t)()) \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_f) = (K, V) \rightarrow W \\
\text{map}[e_m.\text{mapValues}(e_f)] &= \lambda(x : \llbracket K \rrbracket_t). \text{if}(\llbracket e_m \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)(), \\
&\quad \text{Some}(\llbracket e_f \rrbracket [\llbracket e_m \rrbracket [x].\text{value}]), \\
&\quad \text{None}(\llbracket W \rrbracket_t)()) \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_f) = V \rightarrow W \\
\text{map}[e_m.\text{filter}(e_p)] &= \lambda(x : \llbracket K \rrbracket_t). \text{if}(\llbracket e_m \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)() \wedge \llbracket e_p \rrbracket [x, \llbracket e_m \rrbracket [x].\text{value}], \\
&\quad \text{Some}(\llbracket e_m \rrbracket [x].\text{value}), \\
&\quad \text{None}(\llbracket V \rrbracket_t)()) \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_p) = (K, V) \rightarrow \text{bool} \\
\text{map}[e_{m_1}.\text{zip}(e_{m_2})] &= \lambda(x : \llbracket K \rrbracket_t). \text{if}(\llbracket e_{m_1} \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)() \wedge \llbracket e_{m_2} \rrbracket [x] \neq \text{None}(\llbracket W \rrbracket_t)(), \\
&\quad \text{Some}(\text{Tuple_ctor}(\llbracket e_{m_1} \rrbracket [x].\text{value}, \llbracket e_{m_2} \rrbracket [x].\text{value})), \\
&\quad \text{None}(\llbracket \text{Tuple}(V, W) \rrbracket_t)()) \\
&\text{ where } \text{typeof}(e_{m_1}) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_{m_2}) = \text{Map}\langle K, W \rangle \\
\text{map}[e_{m_1}.\text{combine}(e_{m_2}, e_f)] &= \lambda(x : \llbracket K \rrbracket_t). \text{if}(\llbracket e_{m_1} \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)() \wedge \llbracket e_{m_2} \rrbracket [x] \neq \text{None}(\llbracket W \rrbracket_t)(), \\
&\quad \text{Some}(\llbracket e_f \rrbracket [\llbracket e_{m_1} \rrbracket [x].\text{value}, \llbracket e_{m_2} \rrbracket [x].\text{value}]), \\
&\quad \text{if}(\llbracket e_{m_1} \rrbracket [x] \neq \text{None}(\llbracket V \rrbracket_t)(), \\
&\quad \quad \llbracket e_{m_1} \rrbracket [x], \\
&\quad \quad \text{if}(\llbracket e_{m_2} \rrbracket [x] \neq \text{None}(\llbracket W \rrbracket_t)(), \\
&\quad \quad \quad \llbracket e_{m_2} \rrbracket [x], \\
&\quad \quad \quad \text{None}(\llbracket V \rrbracket_t)())) \\
&\quad \text{None}(\llbracket V \rrbracket_t)()) \\
&\text{ where } \text{typeof}(e_{m_1}) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_{m_2}) = \text{Map}\langle K, V \rangle \text{ and } \text{typeof}(e_f) = (V, V) \rightarrow V \\
\text{map}[e_m.\text{toSet}()] &= \lambda(x : \text{Tuple}\langle \llbracket K \rrbracket_t, \llbracket V \rrbracket_t \rangle). \llbracket e_m \rrbracket [x.\text{fst}] = \text{Some}(x.\text{snd}) \\
&\text{ where } \text{typeof}(e_m) = \text{Map}\langle K, V \rangle
\end{aligned}$$

■ **Figure 12** Compiling advanced map operations.

to their value, and provide a traditional interface on top (cf. Figure 4). Note that this encoding of vectors and lists on top of maps is only used when verifying proofs in SMT; when compiling to a target language (e.g. Scala or JavaScript), VeriF_x leverages the language’s built-in vector and list data structures.

C.4 Compilation Example

Figure 13 shows a concrete example of a polymorphic set implemented in VeriF_x and its compiled code in Core SMT. The *MSet* class defines a type parameter *V* corresponding to the type of elements it holds. It also contains one field *set* of type *Set*⟨*V*⟩ and defines a polymorphic method *map* that takes a function $f : V \rightarrow W$ and returns a new *MSet* that results from applying *f* on every element. The compiled Core SMT code defines an ADT *MSet* with one type parameter *V* and one constructor *MSet_ctor*. The constructor defines one field *set* of sort *Array*⟨*V*, *bool*⟩ which is the compiled sort for sets. In addition, a polymorphic *MSet_map* function is defined which takes two type parameters *V* and *W*

```

class MSet[V] (set: Set[V]) {
  def map[W] (f: V => W): MSet[W] =
    new MSet(this.set.map(f))
}

adt MSet⟨V⟩ { MSet_ctor(set : Array⟨V, bool⟩) }
fun MSet_map⟨V, W⟩(this : MSet⟨V⟩,
                  f : Array⟨V, W⟩) : MSet⟨W⟩ =
  MSet_ctor(
    λ(y : W).∃(x : V).this.set[x] ∧ f[x] = y)

```

(a) A polymorphic class in VeriF_x. (b) Compiled Core SMT code.

■ **Figure 13** Example of a polymorphic class in VeriF_x and the compiled Core SMT code.

which correspond to *MSet*'s type parameter and *map*'s type parameter respectively. The function takes two arguments, the object that receives the call and the function *f*. The function's body calls the *MSet* constructor with the result of mapping *f* over the set.

D Implementation and Verification of the Buggy Map CRDT

Section 6.1.1 reported on our experience implementing and verifying the buggy and corrected map CRDTs proposed by Kleppmann [40]. In this appendix, we explain the implementation and verification of the *buggy* map CRDT in detail using code examples. We also discuss the counterexample found by VeriF_x.

■ **Specification 2** The buggy map CRDT algorithm, taken from [40].

```

on initialisation do
  values := {}
end on

on request to read value for key k do
  if ∃t, v. (t, k, v) ∈ values then return v else return null
end on

on request to set key k to value v do
  t := newTimestamp()                                     ▷ globally unique, e.g. Lamport timestamp
  broadcast (set, t, k, v) by causal broadcast (including to self)
end on

on delivering (set, t, k, v) by causal broadcast do
  previous := {(t', k', v') ∈ values | k' = k}
  if previous = {} ∨ ∀(t', k', v') ∈ previous. t' < t then
    values := (values \ previous) ∪ {(t, k, v)}
  end if
end on

on request to delete key k do
  if ∃t, v. (t, k, v) ∈ values then
    broadcast (delete, t) by causal broadcast (including to self)
  end if
end on

on delivering (delete, t) by causal broadcast do
  values := {(t', k', v') ∈ values | t' ≠ t}
end on

```

D.1 Original Specification

The buggy map CRDT is a replicated dictionary storing key-value pairs where the values are regular values (i.e. no nested CRDTs). Specification 2 shows the specification of the buggy map CRDT. It defines a **read** operation to fetch the value associated with a certain

■ **Listing 10** Excerpt from the implementation of the buggy map CRDT in VeriF_x.

```

1  enum MapOp[K, V] { Put(k: K, v: V) | Delete(k: K) }
2  enum MapMsg[K, V] {
3    PutMsg(t: Clock, k: K, v: V) |
4    DeleteMsg(t: Clock, k: K) |
5    NopMsg()
6  }
7  class KMap[K, V](clock: Clock, values: Map[K, Tuple[Clock, V]])
8    extends CmRDT[MapOp[K, V], MapMsg[K, V], KMap[K, V]] {
9    def contains(k: K): Boolean = this.values.contains(k)
10   def get(k: K): V = this.values.get(k).snd
11
12   // Prepare phase for the "put" operation
13   // "put" corresponds to the "set" operation in the specification
14   def preparePut(k: K, v: V) = {
15     val t = this.clock
16     new PutMsg(t, k, v)
17   }
18   // Effect phase for incoming "put" messages
19   def put(t: Clock, k: K, v: V) = {
20     val newClock = this.clock.sync(t)
21     if (!this.values.contains(k) ||
22         this.values.get(k).fst.smaller(t))
23       new KMap(newClock, this.values.add(k, new Tuple(t, v)))
24     else
25       new KMap(newClock, this.values)
26   }
27
28   // Prepare phase for the "delete" operation
29   def prepareDelete(k: K) = {
30     if (this.values.contains(k)) {
31       val t = this.values.get(k).fst
32       new DeleteMsg[K, V](t, k)
33     }
34     else
35       new NopMsg[K, V]()
36   }
37   // Effect phase for incoming "delete" messages
38   def delete(t: Clock, k: K) = {
39     if (this.values.contains(k) && this.values.get(k).fst == t)
40       new KMap(this.clock, this.values.remove(k))
41     else
42       new KMap(this.clock, this.values)
43   }
44
45   override def equals(that: KMap[K, V]) =
46     this.values == that.values
47 }

```


key, and two update operations: `set` and `delete` which assign a value to a key, respectively, delete a certain key. Every operation consists of two parts, a prepare phase (denoted “on request”) that prepares a message to be broadcast to every replica (including itself), and an effect phase (denoted “on delivering”) that applies the incoming message. We briefly explain both update operations:

`set(k, v)`. When preparing a `set` operation that assigns a value v to a key k , the replica generates a new and globally unique timestamp t and broadcasts a `(set, t, k, v)` message. When receiving such a message, the replica checks if it already stores a value for this key. If this is not the case, or if the previous value has a smaller timestamp $t' < t$, then it assigns the incoming value v to the key k , thereby, overriding any previous value. On the other hand, if the previous value has a bigger timestamp, then the incoming `set` message is ignored and the previous value is kept.

`delete(k)`. When preparing a `delete` operation that deletes a key k , the replica fetches the timestamp t at which that key was inserted and broadcasts a `(delete, t)` message. Note that the key itself is not added to the message because `set` always inserts a single key with a unique timestamp, hence, the timestamp t uniquely identifies the key. When receiving a `(delete, t)` message, the replica removes the key that was inserted at timestamp t (if it is still present).

D.2 Implementation in VeriFx

Listing 10 shows the implementation of the buggy map CRDT in VeriFx. Every replica (i.e. every instance of the `KMap` class) maintains a local Lamport clock (consisting of a counter and a replica identifier) and keeps a dictionary that maps keys to timestamped values (i.e. a tuple containing a timestamp and a value). This implementation strategy is slightly different from Spec. 2 but more efficient because a dictionary allows for constant-time lookup, insertion, and deletion. We also extended the `DeleteMsg` such that it not only contains the timestamp t but also the key to be deleted (Line 4). This allows for an efficient implementation of `delete` since the replica knows which key to delete and does not have to loop over the map to find the key whose value has timestamp t .

We override equality - which by default is structural equality - because replicas have different Lamport clocks [43] as our implementation of the clocks keeps a unique replica identifier. Hence, two replicas are considered equal if they have the same values, independent of their clocks. We also renamed the `set` operation to `put`. The remainder of the implementation is a straightforward translation from the specification.

D.3 Verification in VeriFx

After implementing the buggy map CRDT in VeriFx we proceeded to the verification of the map. As explained in Section 6.1.1, VeriFx returned invalid counterexamples because it is not aware of the CRDT’s assumptions which are *implicit* in the design. For instance, VeriFx does not know that replicas have unique IDs nor does it know the relation between a replica’s clock and the values it observed. We need to encode these assumptions explicitly such that VeriFx does not consider cases that cannot occur in practice. To this end, we override the `reachable` and `compatible` predicates (cf. Section 5.1). The former defines which states are reachable (i.e. valid), while the latter defines which replicas are compatible.

Listing 11 shows the implementation of the `reachable` and `compatible` predicates. First, we define a state to be reachable iff every value has a unique timestamp (Line 3 to 6) and all values have a timestamp whose count is smaller than the replica’s local clock (Line 10).

■ Listing 11 Encoding the assumptions of the Map CRDT in VeriF_x.

```

1  override def reachable(): Boolean = {
2    // every value must have a unique timestamp
3    !(exists(k1: K, k2: K) {
4      k1 != k2 &&
5      this.values.get(k1).fst == this.values.get(k2).fst
6    }) &&
7    // All the values in the map must have a timestamp < than our local clock
8    // (since we sync our clock on incoming updates)
9    this.values.values().forall((entry: Tuple[Clock, V]) =>
10   entry.fst.counter < this.clock.counter)
11  }
12  private def noValueFromFuture(r1: KMap[K, V], r2: KMap[K, V]) {
13    r1.values.values().forall((entry: Tuple[Clock, V]) => {
14      val t = entry.fst
15      (t.replica == r2.clock.replica) =>:
16      (t.counter < r2.clock.counter)
17    })
18  }
19  override def compatible(that: KMap[K, V]) = {
20    // replicas have unique IDs
21    (this.clock.replica != that.clock.replica) &&
22    // we have no value from the future of the other replica
23    this.noValueFromFuture(this, that) &&
24    // the other did not observe a value from our future
25    this.noValueFromFuture(that, this) &&
26    // unique timestamps
27    !(exists(k1: K, k2: K) {
28      k1 != k2 && this.values.get(k1).fst == that.values.get(k2).fst
29    }) &&
30    // replicas cannot store different values for the same key and timestamp
31    !(exists(k: K) {
32      val thisTuple = this.values.get(k)
33      val thisTimestamp = thisTuple.fst
34      val thisValue = thisTuple.snd
35      val thatTuple = that.values.get(k)
36      val thatTimestamp = thatTuple.fst
37      val thatValue = thatTuple.snd
38      (thisTimestamp == thatTimestamp) && (thisValue != thatValue)
39    })
40  }

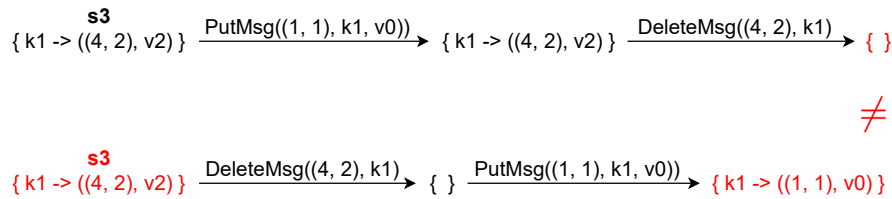
```

```

enum V { v0 | v2 }
enum K { k1 }
val s1 = KMap(Clock(1, 1), Map())
val s2 = KMap(Clock(2, 9), Map(k1 -> (Clock(4, 2), v2)))
val s3 = KMap(Clock(3, 3), Map(k1 -> (Clock(4, 2), v2)))
val x = Put(k1, v0) // operation generated by s1
// The prepare phase will broadcast the following message:
// s1.preparePut(k1, v0) = PutMsg(Clock(1, 1), k1, v0)
val y = Delete(k1) // operation generated by s2
// s2.prepareDelete(k1) = DeleteMsg(Clock(4, 2), k1)

```

(a) Simplified counterexample returned by VeriF_x.



(b) Visualization of the counterexample returned by VeriF_x.

■ **Figure 14** Counterexample for the buggy Map CRDT, found by VeriF_x.

The latter property follows from the fact that the dictionary is constructed by successive insertions and every insertion synchronizes the replica's clock with the timestamp of the inserted element.

Second, we define two replicas to be compatible iff:

- they have unique IDs (Line 21),
- they did not observe values with a timestamp that is bigger than the current clock of the replica that inserted that value (Line 23 to 25) because that would mean that some replica observed a value from the future of the origin replica which is not possible,
- they do not have the same timestamp for different keys (Line 27 to 29) because every insertion inserts a single key with a unique timestamp,
- for every key k for which they store the same timestamp t they also store the same value v (Line 31 to 39) because every timestamp uniquely identifies one insertion: $\text{PutMsg}(t, k, v)$.

Clearly, the above assumptions are not straightforward and are in fact implicit in the original specification, but are nevertheless vital to the correctness of the algorithm. In practice, many CRDTs make similar implicit assumptions which is the reason they are complex and difficult to get right.

Counterexample. After defining all assumptions described above, VeriF_x found a valid counterexample which is shown in Figure 14a. We simplified the counterexample by renaming the keys and values and removing those that do not affect the outcome. The counterexample is equivalent to the one that was found manually by Nair (cf. [40]). It consists of a corner case in which the `Put` and `Delete` operations do not commute and thus may cause replicas to diverge.

Recall that a counterexample is a mapping from variables (defined by the proof) to values that break the proof. In this case, the `CmRDTProof2` trait (cf. Section 5.1) that was used to check commutativity of the operations, defines three variables `s1`, `s2`, and `s3` representing

the state of the replicas, and two variables $x = \text{Put}(k_1, v_0)$ and $y = \text{Delete}(k_1)$ representing concurrent operations that were generated by replica s_1 and s_2 respectively. These replicas first prepare a message for the operations (respectively, $\text{PutMsg}(\text{Clock}(1, 1), k_1, v_0)$) and $\text{DeleteMsg}(\text{Clock}(4, 2), k_1)$) and broadcast those messages to every replica. Every replica receives these messages, possibly in a different order, and applies them.

Depending on the order in which replica s_3 applies the operations, the outcome is different. This is visualized in Figure 14b. If s_3 first processes the $\text{DeleteMsg}(\text{Clock}(4, 2), k_1)$ message then key k_1 is gone because the stored timestamp matches the timestamp that was requested to delete. Afterwards, when processing the $\text{PutMsg}(\text{Clock}(1, 1), k_1, v_0)$ message, the replica will add key k_1 with value v_0 . When applying the operations the other way around, the outcome is different because the $\text{PutMsg}(\text{Clock}(1, 1), k_1, v_0)$ message is ignored since its timestamp is smaller than the timestamp s_3 currently stores for that key: $\text{Clock}(1, 1) < \text{Clock}(4, 2)$. Later, when processing the $\text{DeleteMsg}(\text{Clock}(4, 2), k_1)$ message, s_3 effectively deletes key k_1 because the timestamp matches the one that is stored. Thus, after the first execution, the resulting state contains key k_1 , whereas, after the second execution, k_1 is not present in the map. This explains the divergence bug.

E Verification of the MWS Set

Specification 3 describes the MWS Set, which associates a count to every element. An element is considered in the set if its count is strictly positive. `remove` decreases the element's count, while `add` increments the count by the amount that is needed to make it positive (or by 1 if it is already positive). Listing 12 shows the implementation of the MWS Set in VeriF_x as a polymorphic class that extends the `CmRDT` trait (cf. Section 5.1.3). The type arguments passed to `CmRDT` correspond to the supported operations (`SetOps`), the messages that are exchanged (`SetMsgs`), and the CRDT type itself (`MWSSet`). The `SetOp` enumeration defines two types of operations: `Add(e)` and `Remove(e)`.

The `MWSSet` class has a field, called `elements`, that maps elements to their count (Line 3). Like all op-based CRDTs, the `MWSSet` implements two methods: `prepare` and `effect`. The `prepare` method pattern matches on the operation and delegates it to the corresponding source method which prepares a `SetMsg` to be broadcast to all replicas. The class overrides the `enabledSrc` method to implement the source precondition on `remove`, as defined by Spec. 3. When replicas receive incoming messages, they are processed by the `effect` method which delegates them to the corresponding downstream method which performs the actual update. For example, the `removeDownstream` method processes incoming `RmvMsgs` by decreasing some count k' by 1. Unfortunately, k' is undefined in Spec. 3.

We believe that k' is either defined by the source replica and included in the propagated message (Spec. 4), or, k' is defined as the element's count at the downstream replica (Spec. 5). We implemented both possibilities in VeriF_x (Listings 13 and 14) and verified them to find out which one, if any, is correct. To this end, the companion object of the `MWSSet` class (cf. Line 26 in Listing 12) extends the `CmRDTProof1` trait (cf. Section 5.1.3), passing along three type arguments: the type of operations `SetOp`, the type of messages being exchanged `SetMsg`, and the CRDT type constructor `MWSSet`. The object extends the `CmRDTProof1` trait since the `MWSSet` class is polymorphic and expects one type argument. When executing the proof inherited by the companion object, VeriF_x automatically proves that the possibility implemented by Listing 14 is correct and that the one of Listing 13 is wrong. We thus successfully completed the MWS Set implementation using VeriF_x's integrated verification capabilities.

■ **Specification 3** Op-based MWS Set CRDT taken from Shapiro et al. [67].

```

1: payload set  $S = \{(element, count), \dots\}$ 
2: initial  $E \times \{0\}$ 
3: query lookup (element  $e$ ) : boolean  $b$ 
4:   let  $b = ((e, k) \in S \wedge k > 0)$ 
5: update add (element  $e$ )
6:   atSource ( $e$ ) : integer  $j$ 
7:   if  $\exists (e, k) \in S : k \leq 0$  then
8:     let  $j = |k| + 1$ 
9:   else
10:    let  $j = 1$ 
11:   downstream ( $e, j$ )
12:   let  $k' : (e, k') \in S$ 
13:    $S := S \setminus \{(e, k')\} \cup \{(e, k' + j)\}$ 
14: update remove (element  $e$ )
15:   atSource ( $e$ )
16:   pre lookup( $e$ )
17:   downstream ( $e$ )
18:    $S := S \setminus \{(e, k')\} \cup \{(e, k' - 1)\}$ 

```

■ **Specification 4** Remove with k' defined at source.

```

1: update remove (element  $e$ )
2:   atSource ( $e$ ) : integer  $k'$ 
3:   pre lookup( $e$ )
4:   let  $k' : (e, k') \in S$ 
5:   downstream ( $e, k'$ )
6:    $S := S \setminus \{(e, k')\} \cup \{(e, k' - 1)\}$ 

```

■ **Specification 5** Remove with k' defined in downstream.

```

1: update remove (element  $e$ )
2:   atSource ( $e$ )
3:   pre lookup( $e$ )
4:   downstream ( $e$ )
5:   let  $k' : (e, k') \in S$ 
6:    $S := S \setminus \{(e, k')\} \cup \{(e, k' - 1)\}$ 

```

■ **Listing 12** MWS Set implementation in VeriF_X.

```

1 enum SetOp[V] { Add(e: V) | Remove(e: V) }
2 enum SetMsg[V] { AddMsg(e: V, dt: Int) | RmvMsg(e: V) }
3 class MWSSet[V](elements: Map[V, Int]) extends
4   CmRDT[SetOp[V], SetMsg[V], MWSSet[V]] {
5   override def enabledSrc(op: SetOp[V]) = op match {
6     case Add(_) => true
7     case Remove(e) => this.preRemove(e) }
8   def prepare(op: SetOp[V]) = op match {
9     case Add(e) => this.add(e)
10    case Remove(e) => this.remove(e) }
11   def effect(msg: SetMsg[V]) = msg match {
12     case AddMsg(e, dt) => this.addDownstream(e, dt)
13     case RmvMsg(e) => this.removeDownstream(e) }
14   def lookup(e: V) = this.elements.getOrElse(e, 0) > 0
15   def add(e: V): SetMsg[V] = {
16     val count = this.elements.getOrElse(e, 0)
17     val dt = if (count <= 0) (count * -1) + 1 else 1
18     new AddMsg(e, dt) }
19   def addDownstream(e: V, dt: Int): MWSSet[V] = {
20     val count = this.elements.getOrElse(e, 0)
21     new MWSSet(this.elements.add(e, count + dt)) }
22   def preRemove(e: V) = this.lookup(e)
23   def remove(e: V): SetMsg[V] = new RmvMsg(e)
24   def removeDownstream(e: V): MWSSet[V] = {
25     val kPrime = ??? // undefined in Specification 3
26     new MWSSet(this.elements.add(e, kPrime - 1)) } }
27 object MWSSet extends CmRDTProof1[SetOp, SetMsg, MWSSet]

```

■ **Listing 13** Computing k' at the source.

```

1 def remove(e: V): Tuple[V, Int] =
2   new Tuple(e, this.elements.getOrElse(e, 0))
3 def removeDown(tup: Tuple[V, Int]): MWSSet[V] = {
4   val e = tup.fst; val kPrime = tup.snd
5   new MWSSet(this.elements.add(e, kPrime - 1)) }

```




■ **Listing 14** Computing k' downstream.

```

1 def remove(e: V): V = e
2 def removeDown(e: V): MWSSet[V] = {
3   val kPrime = this.elements.getOrElse(e, 0)
4   new MWSSet(this.elements.add(e, kPrime - 1)) }

```


On Leveraging Tests to Infer Nullable Annotations

Jens Dietrich   

Victoria University of Wellington, New Zealand

David J. Pearce  

ConsenSys, Wellington, New Zealand

Mahin Chandramohan 

Oracle Labs, Brisbane, Australia

Abstract

Issues related to the dereferencing of null pointers are a pervasive and widely studied problem, and numerous static analyses have been proposed for this purpose. These are typically based on dataflow analysis, and take advantage of annotations indicating whether a type is nullable or not. The presence of such annotations can significantly improve the accuracy of null checkers. However, most code found in the wild is not annotated, and tools must fall back on default assumptions, leading to both false positives and false negatives. Manually annotating code is a laborious task and requires deep knowledge of how a program interacts with clients and components.

We propose to infer nullable annotations from an analysis of existing test cases. For this purpose, we execute instrumented tests and capture nullable API interactions. Those recorded interactions are then refined (sanitised and propagated) in order to improve their precision and recall. We evaluate our approach on seven projects from the spring ecosystems and two google projects which have been extensively manually annotated with thousands of `@Nullable` annotations. We find that our approach has a high precision, and can find around half of the existing `@Nullable` annotations. This suggests that the method proposed is useful to mechanise a significant part of the very labour-intensive annotation task.

2012 ACM Subject Classification Software and its engineering → Software defect analysis; Software and its engineering → Software reliability; Software and its engineering → Dynamic analysis

Keywords and phrases null analysis, null safety, testing, program analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.10

Supplementary Material *Software (Source Code):*

<https://github.com/jensdietrich/null-annotation-inference>

archived at `swh:1:dir:af57d8b58579b09bdab080493b944d0a325821ed`

Funding *Jens Dietrich:* The first author was supported by Oracle Labs, Australia.

Acknowledgements The authors would like to thank Chris Povirk for his feedback on using our tool on *guava*, and Görel Hedin for assisting us to set up the experiment reported in Section 7.9.

1 Introduction

Null-pointer related issues are one of the most common sources of program crashes. Much research has focused on this issue, including: eliminating the problems of `null` in new language designs [55, 48, 51, 58]; mitigating the impact of `null` in existing programs [23, 66, 5, 19]; and, developing alternatives for languages stuck with `null` [20, 29, 67].

More recently, several industrial-strength static analyses have been developed to operate at scale, such as *infer* / *nullsafe* [1, 19] and *nullaway* [5]. Such tools employ some form of dataflow analysis and take advantage of an extended type system that distinguishes in some way between nullable and nonnull types [23]. Here, a nonnull type is considered a subtype of a nullable type, and this relationship enables checkers to identify illegal assignments pointing to potential runtime issues. In Java, the standard annotation mechanism can be



© Jens Dietrich, David J. Pearce, and Mahin Chandramohan;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 10; pp. 10:1–10:25

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 On Leveraging Tests to Infer Nullable Annotations

used to define such custom *pluggable* types [9]. For instance, using an annotation defined in JSR305 (i.e., the `javax.annotation` namespace), we can distinguish between the two types `Nullable String` and `Nonnull String`, with `Nonnull String` being a subtype of `Nullable String`. In a perfect world, developers would annotate all methods and fields, allowing static checkers to perform analyses with high recall and precision. Not surprisingly, this hasn't happened. Annotating code is generally a complex problem [13], and recent developer discussions reflect this. For instance, for *commons-lang* the issue LANG-1598 has been open since 14 August 20.¹ In a comment on this issue one developer commented “Agreed this idea, but it is a HUGE work if we want to add `NotNull` and `Nullable` to all public functions in *commons-lang*.” A similar comment can be found in a discussion on adding null-safety annotations to *spring boot* (“it may well be a lot of work”).²

Null-related annotations form part of a contract between the provider and consumer of an API. For instance, consider a library that provides some class `Foo` with a method `String foo()`. Adding an annotation may change this to `Nullable String foo()`. This alters the contract with downstream clients which may have assumed the return was not nullable. Technically this change weakens the postcondition, thus violating Liskov's Substitution Principle (LSP) [41].³ This may therefore cause breaking changes, forcing clients to refactor, for instance, by guarding call sites to protect against null pointer exceptions. Such a change may imply the downstream client was using the API incorrectly (i.e. by assuming `null` could not be returned). As such, one might argue the downstream client is simply at fault here and this change helps expose this. But, such situations arise commonly and oftentimes for legitimate reasons: perhaps the downstream client uses the API in such a way that, in fact, `null` can never be returned; or, the method in question only returns `null` in very rare circumstances which weren't triggered despite extensive testing by the downstream client. Regardless, developers must gauge the impact of such decisions carefully when modifying APIs. This illustrates the complexity of the task, and suggests that it is laborious and therefore expensive to add nullability-related annotations to projects.

Null checkers deal with missing annotations by using defaults to fill in the blanks. Those assumptions have a direct impact on recall and precision. The question arises whether suitable annotations can be inferred by other means.⁴ Indeed, some simple analyses could be used here in principle, such as harvesting existing runtime contract checks. Using such checks is increasingly common as programmers opt to implement defensive APIs in order to reduce maintenance costs [17]. This includes the use of contract APIs such as *guava's Preconditions*⁵, *commons-lang3's Validate*⁶, *spring's Assert*⁷ and the standard library `Objects::requireNonNull` protocol which all include non-null checks. Such an analysis could boost the accuracy of static null checkers that integrate with the compiler, as those contract APIs are defined in libraries that are usually outside the scope of the analysis performed by static checkers. However, exploiting the call sites of such methods is of limited benefit as those checks would only establish that a reference *must not be null*.

¹ Open as of 20 October 22, see <https://issues.apache.org/jira/browse/LANG-1598>

² <https://github.com/spring-projects/spring-boot/issues/10712>

³ LSP was formulated for safe subtyping, but can be applied in this context if we consider evolution as replacement

⁴ Other here means not using the same technique used by static checkers. One could argue that if a static dataflow analysis was used to infer annotations, then that should be integrated into the checker in the first place

⁵ <https://guava.dev/releases/21.0/api/docs/com/google/common/base/Preconditions.html>

⁶ <https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/Validate.html>

⁷ <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/util/Assert.html>

It is much more beneficial for static checkers to annotate code indicating that a reference *may be null* (i.e., “*is nullable*”). The reason is that many static checkers use the *non-null-by-default* assumption that was suggested by Chalin and James after studying real-world systems and finding the vast majority of reference type declarations are not null, making this a sensible choice to reduce the annotation burden for developers [14]. They also point out that this is consistent with default choices in some other languages. The *checkerframework* and *infer* nullness checkers are based on this assumption, whilst some other null checkers such as the one embedded in the Eclipse IDE can be configured as such. Sometime, this is formalised. For instance, the *spring framework* makes the use of the *non-null-by-default* assumption explicit by defining and using two package annotations⁸ `@NonNullApi` and `@NonNullFields` in `org.springframework.lang`, with the following semantics (`@NonNullApi`, similar for `@NonNullFields` for fields): “*A common Spring annotation to declare that parameters and return values are to be considered as non-nullable by default for a given package*”.⁹

Using dynamic techniques is a suitable approach to observe nullability, and can be combined with static analyses to improve accuracy. Such hybrid techniques consisting of a dynamic pre-analysis feeding into a static analysis have been used very successfully in other areas of program analysis [7, 31]. A common reason to use those approaches is to boost recall [65].

In this paper, we explore this idea of inferring nullable annotations from test executions. This is based on the assumption that tests are a good (although imperfect) representation of the intended semantics of a program. We then refine those annotations by means of various static analyses in order to reduce the number of both false positives and false negatives.

Our analysis is *sound* in the sense that it will not infer or add `@NonNull` to a method or field where it may become inconsistent with the runtime behaviour of the program. It is *conservative* in the sense that it will never retract `@Nullable` annotations added by developers.

This paper makes the following contributions:

1. **a dynamic analysis** to capture nullable API interactions representing potential `@Nullable` annotations (“nullability issues”) from program executions,
2. **a set of static analyses (“sanitisation”)** to identify false positives
3. **a static analysis (“propagation”)** to infer additional nullability issues from existing issues
4. **a method to mechanically add the annotations inferred** into projects by manipulating the respective abstract syntax trees (ASTs)
5. **an experiment** evaluating how the annotations we infer compare to existing `@Nullable` annotations of seven projects in the spring framework ecosystem and two additional google projects, containing some of the most widely used components in the Java ecosystem
6. **an open source implementation** of the methods and algorithms proposed, available from <https://github.com/jensdietrich/null-annotation-inference>

These contributions directly relate to concrete research questions which we study in the context of evaluation experiments in Section 7.

Our approach meets the expectations of engineers for tools with high precision [6, 59], and has clear economic benefits – it can partially automate the expensive task of manually annotating code. At the time of writing, some of the results produced by our tool have already been adapted into *spring* and *guava*.

⁸ I.e., annotation used in `package-info.java`

⁹ <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/lang/NonNullApi.html>

10:4 On Leveraging Tests to Infer Nullable Annotations

The paper is organised as follows. Starting with the introduction in this section, we provide a high-level overview of our approach in Section 2. This is followed by a more detailed discussion of the major steps of our method: the capture of potential nullability issues from the execution of instrumented tests (Section 3), the removal of likely false positives through *sanitisation* (Section 4), and inference of additional issues to address false negatives through *propagation* (Section 5). We then briefly discuss a utility to inject the annotations our tool infers back into programs in Section 6, and present evaluation experiments in Section 7. This includes the formulation of four research questions in Section 7.3. Finally, we discuss related work in Section 8, and finish with a brief conclusion in Section 9.

2 Approach

Our approach consists of the following steps and the construction of a respective processing pipeline:

1. **Capture:** The execution of an instrumented program and the recording of *nullability issues*, i.e. uses of `null` in method parameters, returns and fields.
2. **Refinement:** The refinement of nullability issues captured using several light-weight static analyses.
 - a. **Sanitisation:** The identification and removal of nullability issues captured that may not be suitable to infer `@Nullable` annotations to be added to the program, therefore eliminating potential false positives.
 - b. **LSP Propagation:** The inference of additional nullability issues to comply with Liskov’s Substitution Principle [41], therefore addressing potential false negatives.
3. **Annotation:** the mechanical injection of captured and inferred annotations into projects.

These steps are described in detail in the following sections.

3 Capture

3.1 Driver Selection

A dynamic analysis can be used to observe an executing program, and to record when `null` is used in APIs that can then be annotated. The question arises which driver to use to exercise the program. One option is to use existing tests, assuming they are representative of the expected and intended program behaviour.

If libraries are analysed there is another option – to use the tests of *downstream clients*. This approach has been shown to be promising recently to identify breaking changes in evolving libraries [46]. The advantage is that clients can be identified mechanically using an analysis of dependency graphs exposed by package managers and the respective repositories.¹⁰ However, this raises the question which clients to use. Using an open world assumption to include all visible clients (i.e., excluding clients not in public repositories) is practically impossible given the high number of projects using commodity libraries like the ones we have in our dataset. There is no established criteria of how to select *representative* clients.

¹⁰Note that this requires the analysis of incoming dependencies, which is not as straightforward as the analysis of outgoing dependencies (which can simply use the maven dependency plugin) and requires some manual analysis, web site scraping or use of third-party repository snapshots such as *libraries.io*

In principle, synthesised tests [53, 28] could also be used. However, they expose *possible*, but not necessarily *intended* program behaviour. Using synthesised tests would therefore likely result in too many `@Nullable` annotations being inferred. We note that some manually written tests may have the same issue. We will address this in Section 4.2.

In the approach presented here we opted to use only a project’s own tests for generating actual annotations.

3.2 Instrumentation

In order to instrument tests, Java agents were implemented to record uses of `null` in APIs during the execution of tests. These agents can be deployed by modifying the (Maven or Gradle) build script of the project under analysis. The agents intercept code executions using the following six rules which check for occurrence of null references during program execution, and record those occurrences:

ARG at method entries, parameter values are checked for `null`

RET at method exits, return values are checked for `null`

FL1 at constructor (`<init>`) exits, reflection is used to check non-static fields for `null`

FL2 at non-static field writes (i.e. the `putfield` bytecode instruction), the value to be set is checked for `null`

SFL1 at class initialiser (`<clinit>`) exits, reflection is used to check static fields for `null`

SFL2 at static field writes (i.e. the `putstatic` bytecode instruction), the value to be set is checked for `null`

We have implemented agents implementing those rules using a combination of *asm* [11] and *bytebuddy* [70]. If `null` is encountered, *nullability issues* are created and made persistent (written to disk).

Instrumentation can be restricted to certain (project-specific) packages, a system variable is used to set a package prefix for this purpose. This is to filter out relevant issues early as the amount of data collected is significant (see results in Table 2, column 3).

3.3 Capturing Context

A nullability issue is identified by the position of the nullable API element (return type or argument index), and the coordinates (class name, method name, descriptor) of the respective method or field. We are also interested to capture and record the execution context for several reasons:

1. to record sufficient information providing provenance about the execution, sufficient for an engineer who has to decide whether to add a `@Nullable` annotation or not
2. related to the previous item, the number of contexts in which a nullable issue has been observed may itself serve as a quality indicator for the issue – more observed contexts provide some support for this being an issue (instead of a single tests triggering “unintended” program behaviour)
3. to distinguish issues detected by running a project’s own tests from issues detected by running client tests
4. to facilitate the sanitisation of issues, with some sanitisation techniques analysing the execution context.

In order to achieve this, we record the stack during capture. From the stack, we can then infer the *trigger*, i.e. the test method leading to the issue. The following algorithm is used to remove noise from the captured stack and identify the trigger:

10:6 On Leveraging Tests to Infer Nullable Annotations

1. the invocation of `java.lang.Thread::getStackTrace` triggering the stacktrace capture is removed from the stacktrace
2. all elements related to the instrumentation are removed
3. elements related to test processing (*surefire*, *junit*), reflection and other JDK-internal functionality are removed based on the package names of the respective classes owning those methods ¹¹
4. the last element in the stacktrace is set to be the trigger

3.4 Example

Listing 1 shows an issue captured running a test in *spring-core* and serialized using JSON. The test (trigger) is `ConcurrentReferenceHashMapTests::shouldGetSize`, it uses the `Map::put` API implemented in `ConcurrentReferenceHashMap`, which leads to `put` returning `null`.

■ **Listing 1** A serialised nullability issue captured in *spring-core* (for better readability `org.springframework.util` is replaced by `$$`).

```
1 {
2   "className": "$$.ConcurrentReferenceHashMap",
3   "methodName": "put",
4   "descriptor": "(Ljava/lang/Object;Ljava/lang/Object;Z)Ljava/lang/Object;",
5   "kind": "RETURN_VALUE",
6   "argsIndex": -1,
7   "stacktrace": [
8     "$$.ConcurrentReferenceHashMap::put:282",
9     "$$.ConcurrentReferenceHashMap::put:271",
10    "$$.ConcurrentReferenceHashMapTests::shouldGetSize:331"
11  ]
12 }
```

3.5 Deduplication

When issues are captured, it is common that several versions of the same issue are being reported. For instance, there might be two nullability issues reported for the return type of the same method in the same class, but triggered by different tests, and therefore with different stack traces. Throughout the paper, only deduplicated (aggregated) issue counts are reported unless mentioned otherwise. The raw issues might still be of interest as they differ with respect to their provenance, which might be important for a developer reviewing issues.

3.6 Limitations

Our approach does not support generic types. For instance, consider a method returning `List<String>`. In order to establish that the list may contain `Nullable` strings the analysis would need to traverse the object graph of the list object using reflection or some similar method, in order to check that some elements of the list are (or in general some referenced objects associated with the type parameters) are nullable. This is generally not scalable.

Secondly, there are dynamic programming techniques that may bypass the instrumentation. This is in particular the case if reflective field access is used, either directly using reflection, or via deserialisation. This is a known problem, however, reflective field access is rare in practice [65].

¹¹ More specifically, we consider methods in packages starting with the following prefixes as noise: `java.lang.reflect.`, `org.apache.maven.surefire`, `org.junit.`, `junit.`, `jdk.internal`.

4 Sanitisation

4.1 Scope Sanitisation

When exercising code using instrumented tests, potential issues are captured and recorded for all classes including classes defined in dependencies, system and project classes. By setting project-specific namespace (package) prefixes, the analysis can be restricted to project-defined classes only as discussed in Section 3.2. However, this still does not distinguish between classes used at runtime (in Maven and Gradle, this is referred to as the *main* scope), and classes only to be used during testing (the *test* scope). Engineers may not see the need to annotate test code, and a static null checker would usually be configured to ignore test code as its purpose is to predict runtime behaviour such as potential null dereferences resulting in runtime exception.

The analysis to filter out classes not defined in main scope is straightforward: scopes are encoded in the project directory structure if build systems like Maven and Gradle are used. Those build systems and the associated project structures are the defacto-standards used in Java projects [2]. For instance, *spring* uses Gradle, and the compiled classes in main scope can be found in `build/classes/java/main`. The main scope sanitiser simply removes issues in classes not found in this folder.

4.2 Negative Test Sanitisation

The code in Listing 2 from the *spring-core* project is an example of a defensive API practice in `org.springframework.util.Assert`. A runtime exception is used to signal a violated pre-condition, a null parameter in this case. The exception (`IllegalArgumentException`) is thrown in the `Assert::notNull` utility method. While a null pointer exception is also a runtime exception, throwing an `IllegalArgumentException` here is more meaningful as this is (expected to be) thrown by the application, not by the JVM, and clearly communicates to clients that this is a problem caused by how an API is used, as opposed to an exception caused by a bug within the library.

■ **Listing 2** A defensive API in *spring-core*, `org.springframework.util.Assert::isInstanceOf`.

```
1 public static void isInstanceOf(Class<?> type, @Nullable Object obj, String message) {
2     assertNotNull(type, "Type to check against must not be null");
3     ..
4 }
```

This contract is then tested in `org.springframework.util.AssertTests::isInstanceOfWithNullType`, shown in Listing 3.

■ **Listing 3** Testing a defensive API in *spring-core* with JUnit5.

```
1 @Test void isInstanceOfWithNullType() {
2     assertThrows(IllegalArgumentException.class, () -> Assert.isInstanceOf(
3         null, "foo", "enigma")
4     ).withMessageContaining(..);
5 }
```

We refer to such tests as *negative tests* – i.e. tests that exercise abnormal and unintended but possible behaviour, and use an exception or error as the test oracle for this purpose. Features often used to implement such tests are the `assertThrows` method in JUnit5, and the `expected` attribute of the `@Test` annotation in JUnit4.

Including such tests (as drivers) is likely to result in false positives – nulls are passed to the test to trigger defense mechanisms, such as runtime checks. We therefore excluded issues triggered by such tests. This is done by a lightweight ASM-based static analysis that checks

10:8 On Leveraging Tests to Infer Nullable Annotations

for the annotations and call sites indicating the presence of an exception oracle and produces a list of negative tests, and a second analysis that cross-references the context information captured while recording issues against this list, and removes issues triggered by negative tests.

The analysis checks for the above-mentioned negative test patterns in JUnit4 and JUnit5, and a similar pattern in the popular *assertj* library. We also check for call sites for the JUnit4 and JUnit5 `fail` methods in `try` blocks, usually indicating that tests pass if they enter the corresponding `catch` block. Finally, the analysis looks for call sites of methods in `com.google.common.testing.NullPointerTester`. This is a utility that uses reflection to call methods with null for parameters not marked as nullable, expecting a NPE or an `UnsupportedOperationException` being thrown. This may be considered as over-fitting as *guava* is also part of our data set used for evaluation. However, like JUnit, *guava* is a widely used utility library, which warrants supporting this features in a generic tool.

4.3 Shaded Dependency Sanitisation

Shading is a common practice where library classes and often entire package or even libraries are inlined, i.e. copied into the project and relocated into new name spaces. A common use case is to avoid classpath conflicts when multiple versions of the same class are (expected to be) present in a project [69].

For instance, the return type of `org.springframework.asm.ClassVisitor::visitMethod` is not annotated as nullable. The problem here is that *spring-core* also defines several subclasses of this class overriding the method (including `SimpleAnnotationMetadataReadingVisitor`, package name omitted for brevity), which mark the return type as nullable. Reading this as pluggable types with the non-null by default assumption, with `@Nullable MethodVisitor` being a subtype of `MethodVisitor`, this violates Liskov's substitution principle [41] as the postcondition of a non-null return value is weakened in the overriding method. The reason that engineers wont add the annotation is that this class originates from a shaded dependency.¹² This is usually not done manually, but automated using build plugins such as Maven's *shade plugin*¹³. The respective section of the Gradle build script for *spring-core* is shown in Listing 4.

■ **Listing 4** Shading spec in *spring-corespring-core.gradle*.

```
1 task cglibRepackJar(type: ShadowJar) {
2   archiveBaseName.set('spring-cglib-repack')
3   archiveVersion.set(cglibVersion)
4   configurations = [project.configurations.cglib]
5   relocate 'net.sf.cglib', 'org.springframework.cglib'
6   relocate 'org.objectweb.asm', 'org.springframework.asm'
7 }
```

This makes adding `@Nullable` annotations for those classes unattractive – any developer effort to add them is wasted as the source code is replaced during each build, and modern projects are rebuilt often, sometimes multiple times a day. A possible solution would be to add annotations during code generation at build time, but to the best of our knowledge, there are no suitable tools or meta programming techniques readily available to engineers that could be used for this purpose.

Another option would be to add the annotation to the respective method provided by the shaded library, however, engineers are usually not in a position to make such a change.

A sanitiser to take this into account takes a list of packages corresponding to shaded classes as input, and removes issues detected within those classes.

¹² See <https://github.com/spring-projects/spring-framework/pull/28852> for discussion

¹³ <https://maven.apache.org/plugins/maven-shade-plugin/>

4.4 Deprecation Sanitisation

The final sanitiser removes issues collected from deprecated (i.e., annotated with `@java.lang.Deprecated`) fields, methods or classes. The rationale is that given the significant cost of annotating code, engineers might be reluctant to add annotations to code scheduled for removal, and will consider the inference of such annotations less useful. Such a sanitiser can be implemented with a straightforward byte code analysis as `@Deprecated` annotations are retained in byte code. We used *asm* for this purpose in our implementation.

5 Propagation

Annotating an API with `@Nullable` annotations changes the expectations and guarantees of the API contract with clients. In terms of Liskov’s Substitution principle (LSP), adding `@Nullable` to a method (i.e., to the type it returns) weakens its postconditions if we consider `@NonNull` to be the baseline. To preserve LSP, a non-null return type must not be made nullable in a overriding method.

For nullable arguments, the direction changes: while overriding a method making arguments nullable complies to LSP as expectations (for callers) are weakened, nullable arguments must not be made non-null in overridden methods. If we assume `@NonNull` to be the default, this implies that `@Nullable` should also be applied to the arguments of the overriding method. However, the standard Java language semantics only supports covariant return types (e.g., methods can be overridden using a more specific return type), while for argument types invariance is used. Different null checkers and other languages use a variety of approaches here [13] and it is not completely clear what the canonical approach should be. Therefore, in our proof-of-concept implementation, LSP propagation can be customised to propagate nullability for arguments, or not, with propagation being the default strategy.

Listing 5 illustrates our approach. Assume we have annotated `B::foo` using observations from instrumented test runs. Then we also have to add `@Nullable` to the return type of the overridden method `A::foo`, and (if argument propagation is enabled) to the sole argument of the overriding method `C::foo`.

Listing 5 Propagation of `@Nullable` to Sub- and Supertypes.

```
1 public class A {
2     public @Nullable Object foo (Object arg) ;
3 }
4 public class B extends A {
5     public @Nullable Object foo (@Nullable Object arg) ;
6 }
7 public class C extends B {
8     public Object foo (@Nullable Object arg) ;
9 }
```

LSP propagation is implemented using a lightweight ASM-based analysis that extracts overrides relationships from compiled classes, and cross-references with with captured issues, creating new issues. For provenance, references to the original parent issues leading to inferred issues are captured as well and stored alongside the (JSON-serialised) inferred issues as a *parent* attribute.

5.1 Limitations

There is a limitation to hierarchy-based propagation – subtype relationships may extend across libraries, and we may infer nullable annotations for classes that are not in the scope of the analysis, and cannot be refactored. While project owners know super types (and can use methods like opening issues or creating pull requests for projects we don’t control), they

10:10 On Leveraging Tests to Infer Nullable Annotations

are not in control of subtypes in an open world, and rely that downstream projects would eventually pick up those annotations through notifications from some static analyses tools checking for those issues.

5.2 Sanitisation vs Propagation Fixpoint

Sanitisation and propagation have opposite effects. Preferably, an algorithm used to refine the initially collected nullability issues would reach a unique fix point where the future application of sanitisation and propagation would not change the set of refined nullability issues. However, such a fixpoint does not exist. Consider for instance a scenario where a shaded class has a method that is overridden and has a nullable return type in the overriding method. Then LSP propagation suggests to also add this to the return of the overridden method in the super class (to avoid weakening the post conditions), while sanitisation suggests not to refactor the shaded class. This is the issue we have observed in *spring-core* and discussed in Section 4.3.

6 Annotation Injection

We implemented a tool to inject the inferred annotations into projects, using the following steps:

1. compilation units are parsed into ASTs using the *javaparser* API [62]
2. for each nullable issue, the respective method arguments, returns or fields are annotated by adding nodes representing the `@Nullable` annotation to the respective AST
3. after the AST for a compilation unit is processed, it is written out as a Java source code file
4. if necessary, the respective import for the nullable annotation type used is added to the `pom.xml` project file

The tool has been evaluated using standard JUnit unit tests, and by round-tripping (removing and then reinserting existing annotations) the spring projects studied.

There are different annotation libraries available defining nullable annotations, and static checkers often support multiple such annotations. For this reason, the annotator tool supports pluggable annotations. This abstraction is implemented as a `NullableAnnotationProvider` service, implementations provide the nullable type and package names, and the coordinates of an Maven artifact providing the respective annotation. The default implementation is based on JSR305. Alternative providers can be deployed using the standard Java service loader mechanism.

7 Evaluation

Our evaluation is based on a study of some of the popular real-world projects which have been manually null-annotated by project members. We compare those existing annotations with the annotations captured and inferred by our method, and check those two sets for consistency. This is done by measuring *precision* and *recall*. Informally, those measures represent the ratio of inferred annotations to existing annotations, and the percentage of existing annotations our method is able to infer. More precisely, given a set of existing nullable annotations *Existing* and a set of annotations inferred using our method *Inferred*, we define the following metrics:

$$\begin{aligned}
TP &:= Existing \cap Inferred \\
FP &:= Inferred \setminus Existing \\
FN &:= Existing \setminus Inferred \\
precision &:= |TP| / (|TP| + |FP|) \\
recall &:= |TP| / (|TP| + |FN|)
\end{aligned}$$

Those are standard definitions, however, they need to be used with caution here. The concepts suggest that the existing annotations are *the ground truth*. This hinges on two assumptions:

1. The existing annotations are complete.
2. The project test cases provide enough coverage to exercise all possible nullable behaviour.

The first assumption means that all existing nullable annotations our method fails to infer are in fact false positives. This might not be true as the annotations may not be complete, and we explore this issue further in Section 7.8. Therefore, the precision reported needs to be understood as the *lower precision bound (lpb)* in the sense of false positive detection. The second assumption means that all existing issues our tool cannot detect are false negatives. While this is correct in some sense, it does not necessarily indicate a weakness of our method as such, rather than an issue of the quality of input data, i.e. the quality of tests.

Existing annotations are extracted by using a simple byte code analysis (noting that common nullable annotation use runtime retention), we are looking for `@Nullable` annotations in any package to account for the multiple annotation providers. We also support two semantically closely related annotations defined in widely used utility libraries or tools, *guava's* `@ParametricNullness` and *findbug's* `@CheckForNull`.

7.1 Dataset

The data set we use in our study consists of seven projects (modules) from the *spring framework* ecosystem, plus two additional google projects. Those projects were located by searching the Maven repository for projects using libraries providing `@Nullable` annotations, and the selecting projects that actually use a significant number of those annotations. The reason that we chose this method was that we wanted to use existing annotations as (an approximation of) the ground truth to evaluate the inferred annotations. We were particularly looking for projects backed by large engineering teams and well-resourced organisations, assuming that this would result in high-quality annotations.

Spring is the dominating framework for enterprise computing in Java [68], it is supported by a large developer community, is almost 20 years old and keeps on maintaining and innovating its code base. What makes those projects particularly suitable for evaluation is the fact that they have been manually annotated with `@Nullable` annotations. Spring defines its own annotation for this purpose in *spring-core*¹⁴. The amount of annotations found in those projects is extensive, see Section 7.4 for details. Spring uses gradle as build system.

Spring is organised in modules, projects with their own build scripts producing independent deployable binaries. We selected seven projects with different characteristics in particular with respect to how APIs are provided or consumed: *core*, *beans* and *context* are foundational projects for the spring framework overall, with few dependencies. *orm* and *oxm* are middleware

¹⁴ Defined in in `org.springframework.lang`

10:12 On Leveraging Tests to Infer Nullable Annotations

■ **Table 1** project summary, reporting the number of Java, Kotlin and Groovy source code files for both main and test scope, and branch coverage.

program	version	main			test			coverage
		java	kotlin	groovy	java	kotlin	groovy	
s.-beans	5.3.22	301	2	1	126	4	0	60%
s.-context	5.3.22	640	5	0	483	7	2	63%
s.-core	5.3.22	499	1	0	214	14	0	66%
s.-orm	5.3.22	72	0	0	32	0	0	39%
s.-oxm	5.3.22	31	0	0	19	0	0	58%
s.-web	5.3.22	653	1	0	268	5	0	18%
s.-webmvc	5.3.22	368	3	0	225	5	0	39%
guava	31.1	619	0	0	502	0	0	70%
error-prone	2.18.0	745	0	0	1,222	0	0	73%

components for applications to interact with XML data and relational databases, and integrate with existing frameworks for this purpose like *hibernate*, *jpa* and *jaxb*. Finally, *web* is a utility library for web programming (including an HTTP client), and *webmvc* is a comprehensive application framework based on the model-view-controller design pattern [30].

We also include two additional non-spring programs to demonstrate the generality of the method proposed, and avoid over-fitting for spring. Those are *guava* and *error-prone*, both by google. *Guava* is a very popular utility library, whereas *error-prone* is a code analysis utility, similar to *findbugs*. Those two projects use Maven as build system, and have a modular structure, with some modules only containing tests, test tools or annotations. We analysed nullability for the *errorprone/core* and *guava/guava* modules, respectively.

Table 1 provides an overview of the data set used together with some metrics, broken down by scope as discussed in Section 4.1. While those projects predominately contain Java classes, they also contain a smaller amount of Kotlin and Groovy code. Most of this are tests, and as the capture is based on bytecode instrumentation, those tests are still being used as drivers for the dynamic analysis. The table also contains some coverage data.¹⁵ This provides some indication that the projects detected are well tested, and provide reasonable drivers for a dynamic analysis. The coverage data compares favourably to the coverage observed for typical Java programs [18].

7.2 Capture

For the dynamic analysis, we used the agents described in Section 3. With those agents deployed in the build scripts, ground truth extraction is a matter of running the projects builds using the test targets. The agents collect large amounts of data. For instance, the raw uncompressed size of the nullability issue file collected is 19.96 GB for *spring-context*, 4.11 GB for *guava* and 3.57 GB for *error-prone* (see also Table 2). To avoid memory leaks caused by instrumentation, agents dump data frequently, and after test execution using a shutdown hook.

Not unexpectedly, the presence of the agents significantly prolongs the build times – to around one hour for *spring* and 16 hours for *guava*¹⁶. Profiling reveals that stack capture and IO are performance bottlenecks.

¹⁵Branch coverage is reported, calculated using the *jacoco* coverage tool integrated into the IntelliJ IDEA 2022.2 (Ultimate Edition) IDE, and reporting the values aggregated by IntelliJ for the respective packages

¹⁶Builds were run on a MacBook Pro (16-inch, 2021) with Apple M1 Pro, and OpenJDK 11.0.11

We argue that this is acceptable as this is an one-off effort, i.e. this is not designed to be integrated into standard builds.

7.3 Research Questions

We break down the evaluation into a number of research questions. RQ1 compares the possible nullable annotations collected from instrumented test runs with existing annotations. RQ2 and RQ3 assess the utility of the refinements (sanitisation and propagation) performed on the nullability issues collected to improve recall and precision. Finally, in RQ4 we assess the interaction between sanitisation and propagation.

RQ1 How does nullability observed during test execution compare to existing `@Nullable` annotations?

RQ2 Can sanitisation techniques improve the precision of `@Nullable` annotation inference?

RQ3 Can propagation improve the recall of `@Nullable` annotation inference?

RQ4 Does the repeated application of sanitisation and propagation reach a fixpoint?

Results will be reported in Tables for each RQ, and we will summarise the distribution of recall and lower precision bound values across our dataset at the end of Section 7.7 in Figures 1 (for lpb) and 2 (for recall).

7.4 How does nullability observed during test execution compare to existing `@Nullable` annotations ? [RQ1]

The data to answer this RQ are presented in Table 2. Column 2 (ex) contains the number of `@Nullable` annotations found in the respective program (existing `@Nullable` annotations are extracted and also represented as *extracted issues* to facilitate comparison), column 3 (obs) shows the number of `@Nullable` issues observed during the execution of instrumented tests, corresponding to inferred `@Nullable` annotations. The number of observed issues is surprisingly large, but often, multiple nullability issues are reported for the same field, method parameter or method return. To take this into account, we also report the aggregated issues resulting from deduplication as discussed in Section 3.5 in column 4 (agg), and the aggregation ratio (agg/obs) in column 5. This demonstrates that deduplication is very effective. I.e., nullability reported for a given field, method return or parameter is usually supported by different tests, resulting in different contexts. We see this as a strength of our methods as each context provides independent support for the nullability that is being detected. Finally, we report recall and lower precision bound (r,lpb) in column 6. Both are around 50% with two notable exceptions – the significantly lower recall for *spring-core*, and the significantly lower precision for *spring-context* and *error-prone*.

These results suggests that inferring nullability issues dynamically by only observing tests is not sufficient, and further refinement of those results by means of additional analyses is needed.

7.5 Can sanitisation techniques improve the precision of `@Nullable` annotation inference ? [RQ2]

The various sanitisation techniques discussed in Section 4 address potential false positives. To evaluate their impact, we applied the sanitisers to the observed nullability issues for each program in the data set, and report the number of aggregated inferred nullability issues after sanitisation. We also report the results of applying all sanitisers. The absolute numbers are reported in Table 3, the recall / precision metrics are reported in Table 4.

10:14 On Leveraging Tests to Infer Nullable Annotations

■ **Table 2** RQ1 – existing (ex) vs observed (obs) issues, also reported are the aggregation of observed issues (agg), aggregation ratios (agg/obs) and recall / lower precision bound (r,lpb).

program	ex	obs	agg	agg/obs	r,lpb
s.-beans	1,290	321,851	1,320	0.0041	0.54,0.52
s.-context	1,435	6,872,413	5,945	0.0009	0.49,0.12
s.-core	1,510	175,725	1,171	0.0067	0.52,0.67
s.-orm	377	3,443	279	0.0810	0.47,0.63
s.-oxm	84	501	64	0.1277	0.54,0.70
s.-web	2,025	127,882	1,656	0.0129	0.45,0.55
s.-webmvc	1,437	192,800	2,392	0.0124	0.69,0.41
guava	3,993	2,708,816	4,923	0.0018	0.48,0.39
error-prone	507	1,095,752	1,736	0.0016	0.39,0.11

■ **Table 3** RQ2a – observed issues after applying sanitisers (base – no sanitisation applied, D – deprecation, M – main scope, N – negative tests, S – shading).

program	base	san(D)	san(M)	san(N)	san(S)	san(all)
s.-beans	1,320	1,298	763	1,247	1,320	687
s.-context	5,945	5,922	788	5,662	5,682	718
s.-core	1,171	1,140	999	1,024	1,124	780
s.-orm	279	279	192	270	279	184
s.-oxm	64	64	49	64	64	49
s.-web	1,656	1,606	1,076	1,544	1,656	941
s.-webmvc	2,392	2,374	1,076	2,327	2,392	1,048
guava	4,923	4,813	4,008	3,384	4,923	2,464
error-prone	1,736	1,736	1,337	1,736	1,736	1,337

■ **Table 4** RQ2b – recall and lower precision bound (r,lpb) w.r.t. existing annotations after applying sanitisers (D – deprecation, M – main scope, N – negative tests, S – shading).

program	r,lpb(D)	r,lpb(M)	r,lpb(N)	r,lpb(S)	r,lpb(all)
s.-beans	0.52,0.52	0.54,0.91	0.52,0.53	0.54,0.52	0.50,0.95
s.-context	0.48,0.12	0.49,0.90	0.48,0.12	0.49,0.12	0.47,0.94
s.-core	0.50,0.67	0.52,0.78	0.49,0.72	0.52,0.70	0.47,0.92
s.-orm	0.47,0.63	0.47,0.92	0.45,0.63	0.47,0.63	0.45,0.93
s.-oxm	0.54,0.70	0.54,0.92	0.54,0.70	0.54,0.70	0.54,0.92
s.-web	0.43,0.54	0.45,0.85	0.44,0.57	0.45,0.55	0.42,0.90
s.-webmvc	0.68,0.41	0.69,0.92	0.68,0.42	0.69,0.41	0.67,0.92
guava	0.48,0.40	0.48,0.48	0.48,0.56	0.48,0.39	0.48,0.77
error-prone	0.39,0.11	0.39,0.15	0.39,0.11	0.39,0.11	0.39,0.15

The results suggest that most sanitisers have only a minor impact on precision and, sometimes, those improvements come at the price of slight drops in recall. However, one sanitiser stands out: by focusing on classes in the main scope, the precision can be improved dramatically. This suggests that our instrumented tests pick up a lot of nullability in test classes or other test-scoped classes supporting tests.

After applying all sanitisation techniques, we observe a very high lower precision bound of 0.9 or better for all *spring* programs, with some minor drops in recall. The lower precision bound for *guava* is still fairly high, but surprisingly low for *error-prone*, to be discussed below. Balancing precision and recall is a common issue when designing program analyses, but we believe that the focus should be on precision as developers have little tolerance for false alerts. For instance, it has been reported that “Google developers have a strong bias to ignore static analysis, and any false positives or poor reporting give them a justification for inaction.” [59].

■ **Table 5** Annotated vs annotatable program elements, in the last column the number of annotatable elements of type `java.lang.Void` is reported.

program	annotated	annotatable	annotation ratio	Void usage
s.-beans	1,290	5,230	0.25	0
s.-context	1,435	8,849	0.16	0
s.-core	1,510	10,628	0.14	0
s.-orm	377	1,676	0.22	0
s.-oxm	84	467	0.18	0
s.-web	2,025	13,658	0.15	6
s.-webmvc	1,437	8,317	0.17	1
guava	3,964	25,472	0.16	2
error-prone	507	22,669	0.02	958

To investigate the low lower precision bound we observed for *error-prone* further, we conducted an additional experiment where we calculated the *annotation ratio*. For this purpose, we counted the existing `@Nullable` annotations, and the number of program elements that can be annotated, i.e. fields, method parameters and return types for non-synthetic methods and fields whose type is not a primitive type. The results are displayed in Table 5. This show that the annotation ratio for *error-prone* is by on order of a magnitude lower than for the other programs. Therefore, many of the potential false positives are likely to be true positives, and the existing annotations are not suitable to act as a ground truth here. To investigate the matter further, we looked for patterns amongst the potential false positives detected. One pattern stands out – the frequent use of `java.lang.Void` as method parameter and return type. The respective numbers are shown in Table 5, column 5. The use of `Void` in *error-prone* is unusually high. `Void` has an interesting semantics – this class cannot be instantiated, i.e. *it must be null*, and is therefore always nullable by definition. However, in *error-prone*, the respective method returns and parameters are not annotated as `@Nullable`. Interestingly, this is in violation of one of *error-prone*'s own rule *VoidMissingNullable* (“The type `Void` is not annotated `@Nullable`”) ¹⁷. I.e., *error-prone* is not *dog-fooding* [32] here. *Error-prone* has recently opened an issue to address this ¹⁸. We also note that the *nullaway* checker treats `Void` as nullable ¹⁹, and the *checkerframework* declares `@Nullable` as default for `Void` using a meta annotation ²⁰.

We rerun the recall and precision calculation against a ground truth that interprets `Void` as nullable, and for *error-prone* as expected the result change significantly to a recall of 0.72 and a lower precision bound of 0.79.

After performing sanitisation, we also investigated the context depth, i.e. the size of the stack traces recorded. Without sanitisation this data would be distorted by issues discovered in testing scope, leading to very low context depth. For each aggregated issue equivalence class modulo the deduplication relationship (see Section 3.5), we computed the lowest context depth for all issues in the respective equivalence class, and then counted aggregated issues by this depth. The results are reported in Table 6.

¹⁷ <https://errorprone.info/bugpattern/VoidMissingNullable>

¹⁸ <https://github.com/google/error-prone/issues/3792>

¹⁹ <https://github.com/uber/NullAway/blob/master/nullaway/src/main/java/com/uber/nullaway/NullAway.java>, commit <https://github.com/uber/NullAway/commit/1548c69a27e9e3dd1cb185d04b2e870f3b11a771>

²⁰ <https://checkerframework.org/api/org/checkerframework/checker/nullness/qual/Nullable.html>

10:16 On Leveraging Tests to Infer Nullable Annotations

■ **Table 6** Observed and sanitised issues by context depths.

program	all	2	3	4	5	6	7	8	9	10	>10
s.-beans	687	167	109	64	58	46	35	24	22	39	123
s.-context	718	197	122	76	58	52	25	26	22	11	129
s.-core	780	266	165	105	63	37	32	23	21	10	58
s.-orm	184	23	28	20	18	14	24	2	3	3	49
s.-oxm	49	35	4	1	7	0	0	0	0	0	2
s.-web	941	305	258	149	77	52	10	8	2	9	71
s.-webmvc	1,048	329	195	212	117	50	32	12	9	10	82
guava	2,464	972	606	399	163	122	37	20	13	11	121
error-prone	1,337	8	23	56	4	26	4	8	6	2	1,200

■ **Table 7** RQ3a – effect of propagation, aggregated issue counts and recall / lower precision bound for sanitised issues (s), sanitised and then propagated issues (sp) and sanitised, propagated and re-sanitised issues (sps).

program	s	sp	r,sps	r,lpb(s)	r,lpb(sp)	r,lpb(sps)
s.-beans	687	693	693	0.50,0.95	0.51,0.95	0.51,0.95
s.-context	718	736	736	0.47,0.94	0.48,0.94	0.48,0.94
s.-core	780	791	788	0.47,0.92	0.48,0.91	0.48,0.92
s.-orm	184	184	184	0.45,0.93	0.45,0.93	0.45,0.93
s.-oxm	49	49	49	0.54,0.92	0.54,0.92	0.54,0.92
s.-web	941	949	949	0.42,0.90	0.42,0.90	0.42,0.90
s.-webmvc	1,048	1,059	1,059	0.67,0.92	0.68,0.92	0.68,0.92
guava	2,464	2,503	2,503	0.48,0.77	0.49,0.77	0.49,0.77
error-prone	1,337	1,361	1,361	0.39,0.15	0.43,0.16	0.43,0.16

The results suggest that there are some issues revealed by trivial tests (e.g., tests directly invoking functions with `null` parameters). However, a significant number of issues is revealed by more complex behaviour with deep calling contexts. We consider this to be a strengths of the analysis being presented. Note that the context depths are not inflated by boiler-plate code as the stack traces are cleaned during capture (see Section 3.3).

7.6 Can propagation improve the recall of @Nullable annotation inference ? [RQ3]

Next, we applied propagation to the sanitised nullability issues (using all sanitisers). This can discover additional nullability issues not observable during testing, and therefore improve recall. The results are reported in Table 7. Those results suggests that propagation does not significantly change the quality of the analysis. We observe minor improvements in recall for only four programs in our dataset.

As already discussed in Section 7.5, the results for *error-prone* are heavily impacted by the fact that `Void` is not annotated as nullable. If we consider it as implicitly annotated as nullable, and extend the ground truth used to compare the inferred annotations accordingly, the results change to a recall of 0.73 and a lower precision bound of 0.79. We therefore observe a small increase of the recall for *error-prone* as the result of propagation.

■ **Table 8** RQ3b – number of propagated issues and recall / lower precision bound of propagated issues by type (F – field, P – method parameters, R – method return types).

program	prop(F)	prop(P)	prop(R)	r,lpb(F)	r,lpb(P)	r,lpb(R)
s.-beans	205	279	209	0.81,1.00	0.41,0.90	0.47,0.97
s.-context	308	220	208	0.80,0.98	0.34,0.91	0.41,0.90
s.-core	125	422	241	0.80,1.00	0.43,0.86	0.46,0.97
s.-orm	111	38	35	0.90,1.00	0.21,0.76	0.26,0.89
s.-oxm	35	12	2	0.70,1.00	0.45,0.83	0.00,0.00
s.-web	308	438	203	0.72,0.94	0.36,0.87	0.33,0.91
s.-webmvc	373	319	367	0.95,1.00	0.52,0.87	0.63,0.88
guava	353	1,474	676	0.88,0.98	0.42,0.68	0.48,0.87
error-prone	77	700	584	0.80,0.10	0.47,0.11	0.40,0.23

7.7 Does the repeated application of sanitisation and propagation reach a fixpoint ? [RQ4]

Propagation can introduce new annotations which would otherwise be sanitised, and the process generally does not converge against a fix point. An example was already discussed in Section 5.2. However, it is still a relevant question to study and quantify whether we come close to a fix point, and whether it is common for programs not to reach such a fix point. Therefore, we investigated whether this is a significant observable effect by applying sanitisation to the propagated inferred annotations. This had almost no effect, with only a very few issues in *spring-core* being re-sanitised, the respective data is reported in the columns labelled *sp*s (*sanitised-propagated-sanitised*) in Table 7.

Since propagation is the last step of our inference pipeline (capture-sanitise-propagate), we also report a breakdown of nullability issues by program element annotated, as shown in Table 8. What stands out is that for fields both recall and precision of inferring nullability is better than average.

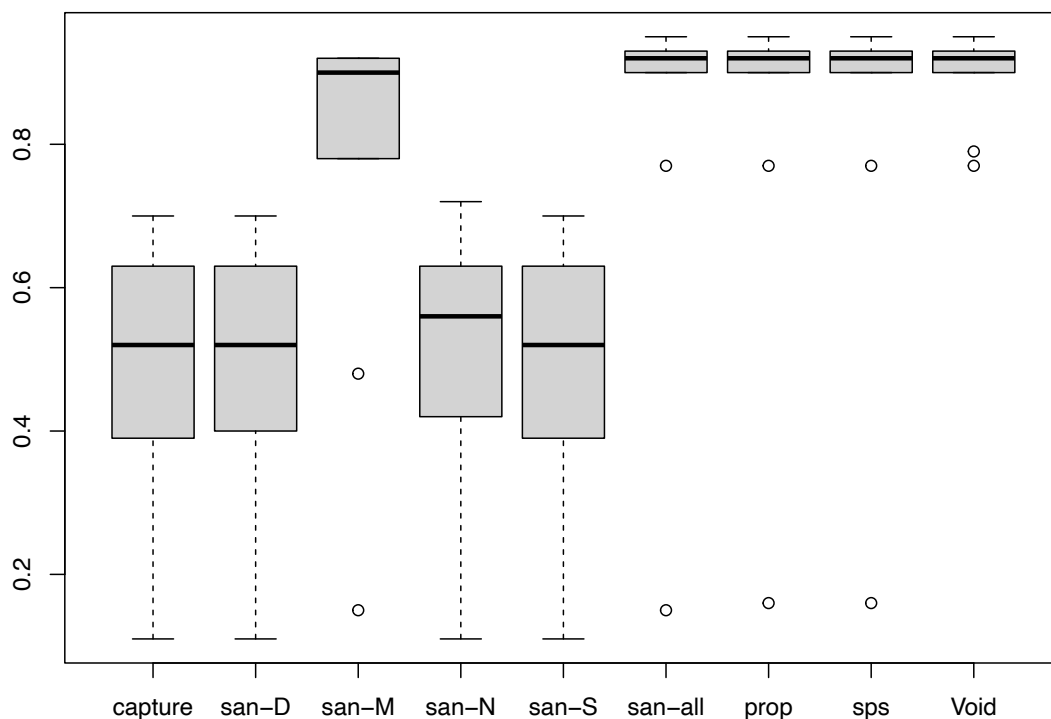
Figures 1 and 2 show the distribution of lpb and recall values across the dataset after each step discussed.

7.8 False False Positives

Despite the generally high precision our approach achieves, it is not perfect. The question arises whether this is caused by false positives. This relates to the fact that our baseline – the existing `@Nullable` annotations, only (under-)approximates the ground truth. In particular, it is unclear whether it is complete. If it was not, some of the false positives our analysis produces would actually be true positives. Sometimes additional analyses can reveal patterns where developers missed annotations that should have been added by some heuristics, an example is the `Void` analysis for *error-prone* discussed in Section 7.5. If no such pattern can be identified, there is another way to find out – add additional annotations inferred by our tool to the respective project(s) via pull requests.

The number of annotations to be added is still relatively large, and given the importance spring has in the developer ecosystem, it can be expected that project owners are generally reluctant to accept pull requests from newcomers. Pull requests have also experienced some amount of inflation recently (partially caused by bots creating pull requests), and therefore processing is delayed.²¹

²¹There were 164 open pull requests on 20 October 2022, <https://github.com/spring-projects/spring-framework/pulls?q=is%3Aopen>



■ **Figure 1** Lower precision bound distribution across the dataset after each step: capture, applying the various sanitisers (san-*), propagation, propagation followed by re-sanitisation (sps), and special handling of Void in *errorprone*.

We have submitted two pull requests with different outcomes: PR1 ²² has resulted in a @Nullable annotation inferred being added ²³. PR2 ²⁴ was rejected, but the developers refined the test the inference is based on ²⁵.

While PR1 and PR2 have resulted in different outcomes, they both have revealed issues in *spring*, and after rerunning the analysis after the action taking by developers in response to the PRs, precision would increase in both cases. Adding an inferred annotation clearly shows that some false positives are actually true positive. Refining the tests has a similar effect – the semantics of tests is sometimes at odds with what is considered intended behaviour, and our tools exposes this. After the test is fixed, the false positive disappears as the tool can no longer infer it.

Our tools has also led to the re-annotation of some classes in *guava* ²⁶.

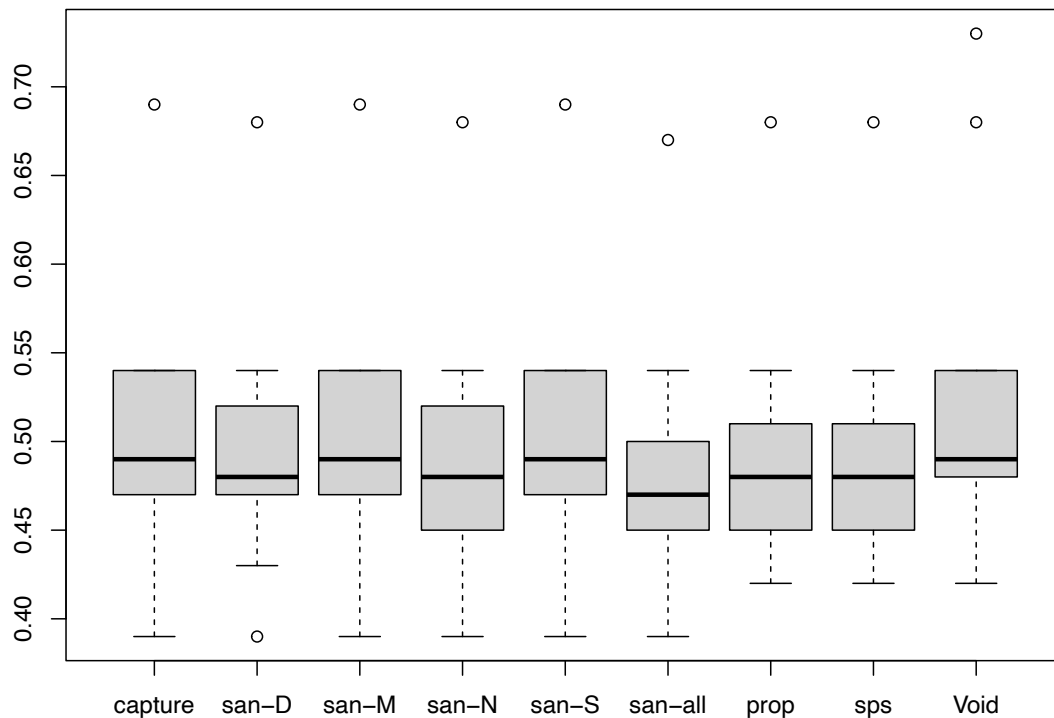
²² <https://github.com/spring-projects/spring-framework/pull/29150>

²³ <https://github.com/spring-projects/spring-framework/commit/35d379f9d3882a02f0368f928b2cecb975404334>

²⁴ <https://github.com/spring-projects/spring-framework/pull/29242>

²⁵ <https://github.com/spring-projects/spring-framework/commit/c14cbd07f449d845269c99faa29241e7e2d0dfc1>

²⁶ <https://github.com/google/guava/commit/2b98d3c1e96b750dc997c29f283084aeb72fb3cf>,
<https://github.com/google/guava/pull/6490>



■ **Figure 2** Recall distribution across the dataset after each step: capture, applying the various sanitisers (san-*), propagation, propagation followed by re-sanitisation (sps), and special handling of Void in *errorprone*.

7.9 Comparison with Purely Static Inference

Houdini [25] infers annotations using the Esc/Java checker. The platform has been deprecated and replaced by other tools, and there is no implementation available. Houdini still uses “pseudo-annotation“ using special markup. This approach is also highly unscalable. The authors report that “*the running time on the 36,000-line Cobalt program was 62 hours*”. For comparison, the version of *spring-core* used in the evaluation experiments alone contains over 146,000 lines of Java code, and checkers rarely scale linearly. For comparison, our analysis generally scales. The bottleneck of our method is the capture, and while this is expensive it generally scales as discussed in Section 7.2.

We contacted the authors of several tools [21, 35, 34] and succeeded in using *jasaddj-nonnullinference* [21] to analyse some programs, and compare results.²⁷ The tool has been maintained until 2015, and based on advice by the authors, we selected some older programs buildable with Java 1.7. The builds had to be heavily customised in order to deal with broken dependencies, details are described in the artefact. The comparison is not straightforward as *jasaddj* infers `@Nonnull` annotations, whereas our method infers `@Nullable`.

The results are shown in Table 9. The *annotatable* column shows the total number of fields, method return and parameters with nullable types. The `@Nonnull` column show the number of annotations inferred by *jasaddj*, and the `@Nullable` columns shows the number of annotations our approach infers. We also report the intersection between both sets in the last column. Both approaches annotate less than half of all annotatable elements. It

²⁷ <https://bitbucket.org/jastadd/jastaddj-nonnullinference>

■ **Table 9** Comparing our approach with JastAddJ NonNull inference.

program	annotatable	@Nonnull	@Nullable	Intersection
commons-lang-3.0	4,647	1,480	1,041	633
commons-cli-3.1	2,724	1,179	65	17
commons-io-2.5	2,241	1,012	326	184
commons-math-3.0	9,404	3,208	270	50

is not clear how to interpret the set complement for both tools. If we interpret everything not `@Nonnull` annotated by *jasaddj* as `@Nullable`, then *jasaddj* has a low precision. The intersection column suggests that there are a significant number of cases where the tools produce inconsistent results. Given the low number of false positive we observe with our tool, it is likely that *jasaddj* produces false positives here.

However, this is not really surprising given that tools like *jasaddj* have been designed to analyse program (as opposed to libraries), where all method calls and field access is known. Our method however is designed for an open world where API interactions from unknown clients have to be considered, and test cases act as proxies for those clients.

8 Related Work

Much work exists on the problem of eliminating null dereferences, of which the vast majority focuses on static checking. Nevertheless, a number of empirical studies exist which are relevant here. The early work of Chalin *et al.* empirically studied the ratio of parameter, return and field declarations which are intended to be non-null, concluding 2/3 are [13, 14]. Another early work was that of Li *et al.* who sampled hundreds of real-world bugs from two large open source projects [40]. They found (amongst other things) null dereferences are the most prevalent of memory-related bugs.

Kimura *et al.* argued that “*it is generally felt that a method returning null is costly to maintain*” [37]. Their study of several open source projects examined whether statements returning `null` or checks against `null` were modified more frequently than others and they observed a difference for the former (but not the latter). Furthermore, they found occurrences of developers replacing statements returning `null` with alternatives (e.g. Null Objects [29] or exceptions) suggesting a desire to move away from using `null` like this. Osman *et al.* also investigated null checks across a large number of open source programs [52]. They found the most common reason developers insert null checks is for method returns and, furthermore, that this is most often to signal errors. The follow-up work of Leuenberger *et al.* investigated the nullability of method returns in Apache Lucene (a widely-used search library) [39]. For each method call site (either internally within Lucene or externally across clients), they identified whether the method return was checked against `null` before being dereferenced (i.e. as this indicates whether the caller expected it could return `null` or not). They confirmed that most methods are expected to return non-null values. However, they also found that external clients were more likely to check a method against `null`, suggesting clients employ defensive behaviour (e.g. when documentation is missing, etc).

8.1 Migration

Dietrich *et al.* harvested lightweight contracts, such as `@Nonnull` and `@Nullable` annotations, from real-world code bases [17]. Unfortunately, they found such annotations are rarely used in practice and that, instead, throwing `IllegalArgumentException` and (to a lesser extent) use of Java `assert` remain predominant. This suggests a key problem faced by all tools for checking non-null annotations (such as those above) is that of annotating existing code bases.

Brotherston *et al.* aimed to simplify migration of existing code bases to use non-null annotations [10]. Their goal is to enable incremental migration of existing code bases to use non-null annotations. Here, developers begin by annotating the most important parts of their system and then slowly widen the net until, eventually, everything is covered. Their approach follows gradual typing [61] and divides programs into the *checked* and *unchecked* portions, such that null dereferences cannot occur in the former. To achieve this, runtime checks are added to unchecked code to prevent exceptions occurring within checked code (i.e. by forcing exceptions at the boundary between them). Such an approach is complementary to our work, and the two could be used together. For example, one might start by inferring annotations using our technique and, subsequently, shift to a gradual typing approach to manage parts where inferred annotations were insufficiently strong, or otherwise require manual intervention. Estep *et al.* further apply ideas of gradual typing to static analysis, using null-pointer analysis as an example [22]. They argue gradual null-pointer analysis hits a “sweet spot” by mixing static and dynamic analysis as needed. A key question they consider is “*why it is better to fail at runtime when passing a null value as a non-null annotated argument, instead of just relying on the upcoming null-pointer exception*”. In essence, they provide two answers: (1) for languages such as C, null dereferences lead to undefined behaviour and, hence, catching them in a controlled fashion is critical; (2) for others, such as Java, it is generally better practice to catch errors as early as possible. Neito *et al.* also take inspiration from gradual typing by considering *blame* across language interop boundaries [50]. In particular, when null-safe languages (e.g. Scala or Kotlin) interact with unsafe languages (e.g. Java), problems can arise.

Houdini statically infers a range of annotations (including non-null) for Java programs [25]. The tool works by generating a large number of candidate annotations and using an existing (modular) checker to eliminate spurious ones. Ekman *et al.* also developed a tool for inferring non-null annotations which could identify roughly 70% of dereferences as safe [21]. Hubert *et al.* formalised an inference tool for non-null annotations based on pointer analysis [35, 34], whilst Spoto developed a similar system arguing it is faster and more precise in practice [63]. XYLEM employs a backwards analysis to find null dereferences [49]. Whilst it doesn’t (strictly speaking) infer annotations, it could be modified to do so. Bouaziz *et al.* also propose a backwards analysis to infer *necessary field conditions* on objects (e.g. that a field is non-null) [8]. This approach is *demand driven* in the sense that fields are marked non-null only if this is necessary to prohibit a null dereference being reported elsewhere.

Finally, inference tools have been developed for pluggable type systems [26, 27, 15, 16]. However, such tools typically cannot account for null checks in conditionals making them relatively imprecise in this context.

8.2 Static Checking

Many tools for statically checking non-null annotations have been proposed. Typically, they differ from traditional type checkers by operating *flow-sensitively* to account for conditional null checks. They also assume non-null annotations have already been added to programs. NULLAWAY provides a nice example here, since it was developed by Uber for static non-null checking at scale [5]. The key requirement was that it could run on all builds, rather than just at code review time (as for a previous tool they used). Their tool is flow-sensitive, but often takes an “optimistic” view (i.e. is unsound). Their reasoning is that sound (i.e. pessimistic) tools produce too many false positives. NULLAWAY does not soundly handle initialisation (see below); likewise, for external (unannotated) code it assumes all interactions are safe. Despite this, they found no cases where unsoundness lead to actual bugs across a 30-day

10:22 On Leveraging Tests to Infer Nullable Annotations

period of usage on a real-world code base. Indeed, this corroborates the earlier findings of Ayewah and Pugh who argued many null dereferences reported by tools do not actually materialise as bugs in practice [4]. As another example, *Eradicate* is part of Facebook Infer [1, 19, 12] and, in many ways, is similar to NULLAWAY.

A number of other tools have been developed which can be used for static `@NonNull` checking, such as FindBugs [33], ESC/Java [24], JastAdd [21], JACK [45] and more [56, 44]. Almost all of these employ flow-sensitive analysis, and many are unsound in various ways (e.g. support for initialisation). Indeed, the initialisation problem has proved so challenging that a large number of works are devoted almost exclusively to its solution [23, 36, 57, 66, 64, 60, 42, 43, 38]. Roughly speaking, the issue is that fields of reference type are assigned a default value of `null` and, thus, every `@NonNull` field initially holds `null` (and this is observable [66]). In our approach we check nullability at the end of object construction. This method is unsound only if super constructors allow access to fields defined in subclasses. We think that this is a rare programming pattern, and note that our approach while aiming for high recall, does not guarantee soundness anyway as it is based on a dynamic analysis.

Finally, so-called “pluggable type systems” [9] allow optional type systems to be layered on existing languages, thus allowing them to evolve independently [26, 27, 15, 3, 16, 47]. The *checkers framework* provides a prominent example which heavily influenced JSR308 (included in Java 8) [54]. A key advantage of this tool over others is the ability to support for flow-sensitive type systems (a.k.a. *flow typing* [55]). Indeed, without this feature checking non-null types is largely impractical [3].

9 Conclusion

We have presented a hybrid analysis pipeline that can be used to capture and refine nullability issues and mechanically inject inferred `@Nullable` annotations into Java programs. Our experiments on some of the most widely used Java commodity libraries demonstrates that this approach is suitable for real-world programs, and that the inferred annotations are consistent with annotations manually added by engineers. In particular, our approach has high precision, and there is evidence from pull requests we have initiated that this precision is potentially higher as our analysis is able to discover missing annotations in the already nullable-annotated programs we have used for evaluation.

Mechanising this process addresses a major issues in real-world projects: the lack of null annotations. Such annotations are part of the program semantics, and generally the annotation process requires deep understanding by project owners and contributors. However, the workload of adding such annotations is significant, and the lack of annotations compromises the utility of static checkers. We have argued that the semantics of which types are nullable and not is already at least partially encoded in existing test cases, and our pipeline exploits this idea of leveraging tests.

The tool has been open sourced and is available from <https://github.com/jensdietrich/null-annotation-inference>.

References

- 1 A tool to detect bugs in Java and C/C++/Objective-C code before it ships. URL: <https://fbinfer.com/>.
- 2 JetBrains Developer Ecosystem Survey 2021, 2021. URL: <https://www.jetbrains.com/lp/devecosystem-2021/java/>.
- 3 C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. OOPSLA*, pages 57–74, 2006.

- 4 Nathaniel Ayewah and William Pugh. Null dereference analysis in practice. In *Proc. PASTE*, pages 65–72. ACM Press, 2010.
- 5 Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. Nullaway: Practical type-based null safety for java. In *Proc. ESEC/FSE'19*, pages 740–750. ACM, 2019.
- 6 Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
- 7 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE'11*, pages 241–250. IEEE, 2011.
- 8 Mehdi Bouaziz, Francesco Logozzo, and Manuel Fähndrich. Inference of necessary field conditions with abstract interpretation. In *Proc. APLAS*, pages 173–189. Springer-Verlag, 2012.
- 9 Gilad Bracha. Pluggable type systems. In *Proc. Workshop on Revival of Dynamic Languages*, 2004.
- 10 Dan Brotherston, Werner Dietl, and Ondrej Lhoták. Granular: gradual nullable types for java. In *Proc. CC*, pages 87–97. ACM Press, 2017.
- 11 Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.
- 12 Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O’Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *Proc. NFM*, pages 3–11. Springer-Verlag, 2015.
- 13 Patrice Chalin and Perry R James. Non-null references by default in java: Alleviating the nullity annotation burden. In *European Conference on Object-Oriented Programming*, pages 227–247. Springer, 2007.
- 14 Patrice Chalin, Perry R James, and Frédéric Rioux. Reducing the use of nullable types through non-null by default and monotonic non-null. *IET Software*, 2(6):515–531, 2008.
- 15 Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proc. PLDI*, pages 85–95. ACM Press, 2005.
- 16 Brian Chin, Shane Markstrum, Todd Millstein, and Jens Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *Proc. ESOP*, 2006.
- 17 Jens Dietrich, David J Pearce, Kamil Jezek, and Premek Brada. Contracts in the wild: A study of java programs. In *Proc. ECOOP'17*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- 18 Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. XCorpus – An executable corpus of Java programs. *Journal of Object Technology*, 16(4):1:1–24, August 2017. doi:10.5381/jot.2017.16.4.a1.
- 19 Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. Scaling static analyses at facebook. *CACM*, 62(8):62–70, 2019.
- 20 Kinga Dobolyi and Westley Weimer. Changing Java’s semantics for handling null pointer exceptions. In *Proc. ISSRE*, pages 47–56. IEEE Computer Society, 2008.
- 21 T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *JOT*, 6(9):455–475, 2007.
- 22 Sam Estep, Jenna Wise, Jonathan Aldrich, Éric Tanter, Johannes Bader, and Joshua Sunshine. Gradual program analysis for null pointers. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 3:1–3:25, 2021.
- 23 Manuel Fähndrich and K Rustan M Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 302–312, 2003.
- 24 C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

- 25 Cormac Flanagan and K Rustan M Leino. Houdini, an annotation assistant for `esc/java`. In *International Symposium of Formal Methods Europe*, pages 500–517. Springer, 2001.
- 26 Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proc. PLDI*, pages 192–203, 1999.
- 27 Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. PLDI*, pages 1–12, 2002.
- 28 Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proc. FSE'11*, pages 416–419, 2011.
- 29 Maria Anna G. Gaitani, Vassilis Zafeiris, N. A. Diamantidis, and Emmanouel A. Giakoumakis. Automated refactoring to the null object design pattern. *Inf. Softw. Technol.*, 59, 2015.
- 30 Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
- 31 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. In *Proc. OOPSLA'17*, pages 1–27. ACM, 2017.
- 32 Warren Harrison. Eating your own dog food. *IEEE Software*, 23(3):5–7, 2006.
- 33 David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proc. PASTE*, pages 13–19. ACM Press, 2005.
- 34 Laurent Hubert. A non-null annotation inferencer for java bytecode. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 36–42, 2008.
- 35 Laurent Hubert, Thomas Jensen, and David Pichardie. Semantic foundations and inference of non-null annotations. In *Proceedings of the International conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 132–149. Springer-Verlag, 2008.
- 36 Laurent Hubert and David Pichardie. Soundly handling static fields: Issues, semantics and analysis. *ENTCS*, 253(5):15–30, 2009.
- 37 Shuhei Kimura, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Does return null matter? In *Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 244–253. IEEE, 2014.
- 38 Alexander Kogtenkov. Practical void safety. In *Proc. VSTTE*, pages 132–151. Springer-Verlag, 2017.
- 39 Manuel Leuenberger, Haidar Osman, Mohammad Ghafari, and Oscar Nierstrasz. Harvesting the wisdom of the crowd to infer method nullness in java. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–80. IEEE, 2017.
- 40 Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33. ACM Press, 2006.
- 41 Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.
- 42 Fengyun Liu, Ondrej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. A type-and-effect system for object initialization. In *Proc. OOPSLA*, pages 175:1–175:28, 2020.
- 43 Fengyun Liu, Ondrej Lhoták, Enze Xing, and Nguyen Cao Pham. Safe object initialization, abstractly. In *Proceedings of the Symposium on Scala*, pages 33–43. ACM Press, 2021.
- 44 Magnus Madsen and Jaco van de Pol. Relational nullable types with boolean unification. In *Proc. OOPSLA*, pages 1–28, 2021.
- 45 C. Male, D.J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for `@NonNull` types. In *Proc. CC*, pages 229–244, 2008.
- 46 Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node.js libraries. In *proc. ECOOP'18*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

- 47 Ana Milanova and Wei Huang. Inference and checking of context-sensitive pluggable types. In *Proc. ESEC/FSE*, page 26. ACM Press, 2012.
- 48 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *PACMPL*, 2(OOPSLA):112:1–112:29, 2018.
- 49 Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *2009 IEEE 31st International Conference on Software Engineering*, pages 133–143. IEEE, 2009.
- 50 Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondrej Lhoták. Blame for null. In *Proc. ECOOP*, pages 3:1–3:28, 2020.
- 51 Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. Scala with Explicit Nulls. In *Proc. ECOOP*, volume 166, pages 25:1–25:26, 2020.
- 52 Haidar Osman, Manuel Leuenberger, Mircea Lungu, and Oscar Nierstrasz. Tracking null checks in open-source java systems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 304–313. IEEE, 2016.
- 53 Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Proc. OOPSLA '07*, pages 815–816. ACM, 2007.
- 54 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proc. ISSTA '08*, pages 201–212. ACM, 2008.
- 55 D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, pages 335–354, 2013.
- 56 P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. CC*, pages 334–554, 2001.
- 57 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proc. POPL*, pages 53–65. ACM Press, 2009.
- 58 Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. Union Types with Disjoint Switches. In *Proc. ECOOP*, volume 222, pages 25:1–25:31, 2022.
- 59 Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspán. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.
- 60 Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix - safe modular circular initialisation with placeholders and placeholder types. In *Proc. ECOOP*, volume 7920, pages 205–229. Springer-Verlag, 2013.
- 61 Jeremy Siek and Walid Taha. Gradual typing for objects. In *Proc. ECOOP*, pages 151–175. Springer-Verlag, 2007.
- 62 Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. Javaparser: visited. *Leanpub*, oct. de, 2017.
- 63 Fausto Spoto. Nullness analysis in boolean form. In *Proc. SEFM*, pages 21–30. IEEE, 2008.
- 64 Fausto Spoto and Michael D. Ernst. Inference of field initialization. In *Proc. ICSE*, pages 231–240. ACM Press, 2011.
- 65 Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *Proc. ICSE'20*, pages 1049–1060. IEEE, 2020.
- 66 Alexander J. Summers and Peter Mueller. Freedom before commitment: A lightweight type system for object initialisation. In *Proc. OOPSLA*, pages 1013–1032, 2011.
- 67 Timothy A. V. Teatro, J. Mikael Eklund, and Ruth Milman. Maybe and either monads in plain C++ 17. In *Proc. CCECE*, pages 1–4. IEEE, 2018.
- 68 Brian Vermeer. Spring dominates the java ecosystem with 60% using it for their main applications, 2020. <https://snyk.io/blog/spring-dominates-the-java-ecosystem-with-60-using-it-for-their-main-applications/>.
- 69 Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. Do the dependency conflicts in my project matter? In *Proc. ESEC/FSE'18*, pages 319–330, 2018.
- 70 Rafael Winterhalter. Byte Buddy – A code generation and manipulation library for creating and modifying Java classes during the runtime, 2014. URL: <https://bytebuddy.net/>.

super-Charging Object-Oriented Programming Through Precise Typing of Open Recursion

Andong Fan  

The Hong Kong University of Science and Technology (HKUST), Hong Kong, China

Lionel Parreaux  

The Hong Kong University of Science and Technology (HKUST), Hong Kong, China

Abstract

We present a new variation of object-oriented programming built around three simple and orthogonal constructs: *classes* for storing object state, *interfaces* for expressing object types, and *mixins* for reusing and overriding implementations. We show that the latter can be made uniquely expressive by leveraging a novel feature that we call *precisely-typed open recursion*. This feature uses “this” and “super” annotations to express the requirements of any given partial method implementation on the types of respectively the current object and the inherited definitions. Crucially, the fact that mixins do *not* introduce types nor subtyping relationships means they can be composed even when the overriding and overridden methods have incomparable types. Together with advanced type inference and structural typing support provided by the MLscript programming language, we show that this enables an elegant and powerful solution to the Expression Problem.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases Object-Oriented Programming, the Expression Problem, Open Recursion

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.11

Related Version *Extended Version*: <https://lptk.github.io/superoop-paper>

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*:
<https://doi.org/10.4230/DARTS.9.2.22>

Software (Open-source implementation): <https://github.com/hkust-taco/superoop>
archived at `swh:1:dir:7446abcf043f3546fae3ebce3efd85c07c70afa3`

Software (Online demonstration): <https://hkust-taco.github.io/superoop>

Acknowledgements We thank the anonymous reviewers, Yaozhu Sun, and Marco Servetto for their helpful comments, as well as Cunyuan Gao for his help with the implementation. This work follows up on concepts previously presented by the first author as a research abstract [10].

1 Introduction

Every object-oriented programming (OOP) developer regularly uses the **super** keyword to access overridden definitions from inherited classes. Yet, this keyword has received relatively little attention in previous OOP literature and has been conspicuously absent from most previous research, with few exceptions [17]. This may be due to the assumption that **super**-calls can be resolved statically and are thus a mere syntactic convenience that is easily desugared into traditional core OOP features [2]. In this paper, we propose to challenge this assumption: noting that **super** is in fact *late-bound* in mixin-composition systems,¹ we describe an OOP approach which assigns *precise types* to **super**-calls to reflect the “open” nature of this late binding. Consider the following prototypical `Point` example class:

```
class Point(x: Int, y: Int)
```

¹ **super** is bound at the time the mixin method where it appears is composed into a class, which can happen as late as runtime in many mixin-composition languages.



11:2 super-Charging Object-Oriented Programming

This class simply defines two coordinates `x` and `y` as immutable fields.

Suppose we want to define a comparison function that works on points. We place this definition in a *mixin* declaration, for reasons that shall soon become clear:

```
mixin ComparePoint {  
  fun compare(lhs: Point, rhs: Point): Bool =  
    lhs.x == rhs.x and lhs.y == rhs.y }
```

Now suppose we want to compare colored points, but we would like colored comparison to be generally specified, so that it can be directly reused with other things than points. This can be done using the following combination of interface and mixin (`Base` is a type parameter):

```
interface Colored { color: String }  
  
mixin CompareColored[Base] {  
  super: { compare: (Base, Base) → Bool }  
  fun compare(lhs: Base & Colored, rhs: Base & Colored): Bool =  
    super.compare(lhs, rhs) && lhs.color.equals(rhs.color) }
```

We define an interface specifying that a `Colored` object should contain a `color` method or field of type `String`. We also define the `CompareColored` mixin, which implements a comparison method *based on* an assumed existing comparison method, inherited from an unknown parent implementation and referred to through **super**. The `Base` type parameter denotes the type compared by that unknown parent implementation; it is needed in order to leave the mixin *open-ended*, i.e., to allow mixing it with arbitrary parent implementations. Notice that the type of `compare` in `CompareColored` is *different* from the one specified in the **super** annotation, and is in particular not a subtype of it: the version defined in `CompareColored` takes parameters of *more precise* type `Base & Colored`, where `&` is an intersection type constructor, meaning that each parameter should be *both* a `Base` *and* a `Colored`. This difference is a crucial ingredient in our precisely-typed open recursion approach.

We now define `ColoredPoint` and place its comparison implementation in a module:²

```
class ColoredPoint(x: Int, y: Int, color: String)  
  extends Point(x, y) implements Colored  
  
module CompareColoredPoint extends ComparePoint, CompareColored[Point]
```

`CompareColoredPoint` did not need to define its own comparison method – that method was *composed automatically* by inheriting from the `ComparePoint` and `CompareColored` mixins, the latter using the correct `Point` base type argument. Note that mixins on the right override those on the left. The signature of `CompareColoredPoint`'s `compare` method, which allows passing in colored points, is:

```
CompareColoredPoint.compare: (Point & Colored, Point & Colored) → Bool
```

which is *not* a subtype of `ComparePoint`'s `compare` method. This is fine because mixins in our approach do not introduce types, and there is thus no subtyping relationship between `CompareColoredPoint` and `ComparePoint`, which is reminiscent of Cook et al. famous assertion that *inheritance is not subtyping* [7].

² A module declares a class with a singleton instance, similar to Scala's **object**.

Now imagine we want to deal with “*nested*” objects, which are objects that may optionally have a parent.³ We can similarly define a comparison mixin for nested objects as follows:

```
interface Nested[A] { parent: Option[A] }

mixin CompareNested[Base, Final] {
  super: { compare: (Base, Base) → Bool }
  this: { compare: (Final, Final) → Bool }

  fun compare(lhs: Base & Nested[Final], rhs: Base & Nested[Final]): Bool =
    super.compare(lhs, rhs) &&
    if lhs.parent is Some(p)
      then rhs.parent is Some(q) and this.compare(p, q)
      else rhs.parent is None
}
```

In this variant, we additionally use a **this** refinement, which specifies the *eventual* types of the methods the current object should support, after all inheritance and overriding is performed. The reason we use **this** and not **super** in the recursive **this.compare(p, q)** call is that we should take into account that *p* and *q* *themselves* may be nested points!

Finally, it is possible to compare points that are *both* nested *and* colored by directly composing the corresponding implementations:

```
class MyPoint(x: Int, y: Int, color: String, parent: Option[MyPoint])
  extends Point implements Colored, Nested[MyPoint]

module CompareMyPoint extends ComparePoint, CompareColored[Point],
  CompareNested[Point & Colored, MyPoint]
```

Or alternatively, in a different order:

```
module CompareMyPoint extends ComparePoint, CompareNested[Point, MyPoint],
  CompareColored[Point & Nested[MyPoint]]
```

Mixin composition order is meaningful because it determines overriding order; moreover, in our approach, the types of methods may change through overriding – here, notice how we pass different type arguments to `CompareColored` and `CompareNested` in each version.

To support this idea of precisely-typed mixin composition, we present the **SuperOOP** system, a simple yet uniquely expressive core description of OOP built around three orthogonal concepts: *classes* for storing object state, *interfaces* for expressing object types, and *mixins* for reusing and overriding implementations.⁴ Notably, we only support inheriting from interfaces and mixins, not from classes.⁵ We show that these simple, orthogonal concepts are sufficient to explain the usual features of object-oriented programming languages, including those with complicated multiple-inheritance disciplines, like Scala’s trait composition approach.

We also describe how the ideas of SuperOOP can be integrated into MLscript, a nascent ML-inspired programming language with structural types and advanced type inference, based on the recently proposed MLstruct type system [26]. Using this approach, all the types can

³ `Option[A]` is defined as the usual algebraic data type, with cases `Some[A](value: A)` and `None`.

⁴ Such separation of concerns was already proposed by previous authors, such as Bettini et al. [2] and Damiani et al. [8], but the systems they developed did not support overriding and open recursion, which is the *raison d’être* of our approach.

⁵ We see in Section 4.1 that the `Point` class inheritance example seen above can be desugared into our core λ^{super} calculus through interface inheritance and without requiring class inheritance.

11:4 super-Charging Object-Oriented Programming

be inferred automatically as long as they do not involve first-class polymorphism (which requires explicit annotations). For instance, in MLscript, the `CompareColored` mixin shown above could also be written as the more lightweight:

```
mixin CompareColored {  
  fun compare(lhs, rhs) =  
    super.compare(lhs, rhs) && lhs.color.equals(rhs.color) }
```

for which our compiler infers the following *mixin signature*:

```
mixin CompareColored:  $\forall$  'A1 'A2 'B . {  
  super: { compare: ('A1, 'A2)  $\rightarrow$  Bool }  
  compare: ('A1 & {color: {equals: 'B  $\rightarrow$  Bool}}, 'A2 & {color: 'B})  $\rightarrow$  Bool }
```

Our specific contributions are summarized as follows:

- We explain the general ideas of SuperOOP in the context of the structurally-typed MLscript programming language, and how it allows solving interesting problems simply and elegantly, including the Expression Problem and derivatives (Section 2). SuperOOP mixins improve on the state of the art by allowing precise typing of open recursion, which to the best of our knowledge was never proposed before.
- We formalize the core concepts of SuperOOP, including its precisely-typed mixin inheritance mechanism, in a declarative type system called λ^{super} . We use big-step semantics to closely reflect a real implementation and prove the soundness of λ^{super} through the preservation and coverage properties (Section 3).
- We discuss the expressiveness and limitations of the presented design of SuperOOP as well as its implementation. We present several important approaches from previous literature on the topic of inheritance and the Expression Problem, and explain how these approaches compare to SuperOOP in detail (Section 4).
- We provide an implementation of MLscript/SuperOOP which demonstrates how type inference can be used to type check concise mixin and class definitions. Both the open-source version and the archived artifact with documentation are available. A demo of this implementation is included in the supplementary material of this paper.⁶

2 Motivation

In this section, we introduce a motivating example in MLscript “**super-charged**” by our OOP approach in more detail.

The Expression Problem and Extensible Variants

In modular programming, the *Expression Problem* (EP) describes the dilemma posed by the modular extension for both data types and their operations in object-oriented and functional programming. There are many ways of tackling this problem, but one of the most straightforward is to rely on some notion of *extensible variants*, as done by Garrigue [16] with OCaml’s polymorphic variants. The general idea of extensible variants is that they are similar to algebraic data types (a.k.a. variants) except that one is able to specify which data type cases are allowed in a given type, and moreover one is able to add new data type cases after the fact.

⁶ The demo is also available at: <https://hkust-taco.github.io/superoop/>.

MLscript supports a simple form of extensible variants implemented through *subtyping and structural types*. In this section, we see how the combination of this feature and SuperOOP's precise typing of open recursion can achieve what we believe is one of the simplest and cleanest solutions to the expression problem so far.

A Quick Look at MLscript

We first take a quick look at MLscript's basic language features that enable a form of extensible variants and serve as key ingredients in our solution to the Expression Problem.

Basic data type classes. Consider the following MLscript class definitions which encode a very minimal expression language that we will later extend in several directions.

```
class Lit(value: Int)
class Add[T](lhs: T, rhs: T)
```

The `Lit` class represents integer literals and the `Add` class represents addition. Note that the types of `Add`'s value parameter are polymorphic, meaning that they can be chosen arbitrarily. We will see that the ability to leave the types of subexpressions open is crucial to the extensibility of our approach.

Union types. Based on these class definitions, we can construct types such as:

```
type LitOrAddLit = Lit | Add[Lit]
```

where `|` is called a *union type* constructor. `LitOrAddLit` represents the type of an expression that is *either* an integer literal *or* an addition between two integer literals.

Equirecursive types. More interestingly, we can define the type of arbitrary expressions in our little language as:

```
type SimpleExpr = Lit | Add[SimpleExpr]
```

Notice that this type is *equirecursive*, meaning that `SimpleExpr` is *equivalent* to its unrolling `Lit | Add[SimpleExpr]`. This is quite convenient in the context of structural typing, and it allows us to have subtyping relationships (denoted as ' $\tau_1 <: \tau_2$ ', meaning that τ_1 is a subtype of τ_2) such as `LitOrAddLit <: SimpleExpr`. An equivalent way of specifying `SimpleExpr` without having to introduce a type declaration is through MLscript's `'as'` binder (similar to `'as'` in languages like OCaml), as in `'Lit | Add['a] as 'a'` (where `'as'` has least precedence).

Evaluation. To use values in our small expression language, we define an `eval` recursive function:

```
fun eval(e) = if e is
  Lit(n)      then n
  Add(lhs, rhs) then eval(lhs) + eval(rhs)
```

This function uses MLscript's syntax for pattern matching, which extends the traditional `if-then-else` syntactic form with multi-way-`if`-style functionality and destructuring through an `'is'` keyword [25]. The type of this function is inferred by MLscript to be:

```
eval: (Lit | Add['a] as 'a) → Int
```

11:6 super-Charging Object-Oriented Programming

Default cases and constructor difference. It is quite instructive to consider what happens when default cases are used in MLscript, as in:

```
fun eval2(e) =
  if e is
    Lit(n)          then n
    Add(lhs, rhs)  then eval2(lhs) + eval2(rhs)
  else e
```

In this case, the type inferred is

```
eval2: (Lit | Add['a] | 'b\Lit\Add as 'a) → (Int | 'b)
```

Above, ‘\’ is a *constructor difference* type operator,⁷ which is used to *remove* concrete class type constructors from a given type (here ‘b’). This type operator applies incrementally, as its left-hand side becomes concretely known upon type instantiation. For instance, after instantiating the type variable ‘b’ to, say, `Add[Int] | Bool` in the type above, ‘b\Lit\Add’ becomes `(Add[Int] | Bool)\Lit\Add`, which is equivalent to just `Bool`. Since all negative occurrences of ‘b’ (here there is only one) are subject to this constructor difference, passing values for ‘b’ which are of the `Lit` or `Add` forms is effectively prevented, which ensures type safety⁸ [26]. On the other hand, any other type constructor is allowed, for example, we could call `eval2(true)`, with inferred result type `Int | Bool`.

Open Recursion in MLscript with SuperOOP Mixins

Now let us consider putting our original evaluation function inside of a *mixin*, in order to enable future extensions. To make the recursion of evaluation *open*, we now recurse through method calls of the form ‘`this.eval`’ (here ‘`this`’ is the class instance to be late-bound) instead of a direct `eval` recursive function call:

```
mixin EvalBase {
  fun eval(e) = if e is
    Lit(n)          then n
    Add(lhs, rhs)  then this.eval(lhs) + this.eval(rhs) }
```

The type signature inferred for that mixin definition is the following:

```
mixin EvalBase: ∀ 'A. {
  this: { eval: ('A) → Int }
  eval: (Lit | Add['A]) → Int
}
```

Above, ‘A’ is a *mixin-level* type variable,⁹ meaning that it must be instantiated to a specific type each time the mixin is inherited as part of a class. Since mixins do *not* introduce types on their own, `EvalBase` cannot be used as a type. Using `EvalBase` as a type would be a

⁷ Constructor difference is not a primitive construct of MLscript’s underlying type system, MLstruct [26]. Type $A \setminus B$ is encoded in that type system as $A \ \& \ \sim\#B$, where $\&$ is the *type intersection* operator, \sim is the *type negation* operator, and $\#B$ represents the *nominal identity* of class B, i.e., its raw type constructor without any fields or type parameters attached.

⁸ Perhaps counter-intuitively, we do not need to restrict the positive occurrences of ‘b’, as they are always effectively unrestricted due to covariance. Consider a function of type $(\text{'b}\setminus\text{Lit}\setminus\text{Add}) \rightarrow \text{'b}$. Substituting `Mul | Lit | Add` for ‘b’ results in $((\text{Mul} \mid \text{Lit} \mid \text{Add})\setminus\text{Lit}\setminus\text{Add}) \rightarrow (\text{Mul} \mid \text{Lit} \mid \text{Add})$, which is equivalent to $\text{Mul} \rightarrow (\text{Mul} \mid \text{Lit} \mid \text{Add})$. This is a supertype of $\text{Mul} \rightarrow \text{Mul}$, which we could have obtained from substituting `Mul` for ‘b’ in the first place, so this type would have been reachable even after a “properly restricted” substitution of ‘b’. In other words, it does not make much sense to restrict the positive occurrence of ‘b’ and there is no practical need for it.

⁹ We use uppercase names for *mixin-level* type variables and lowercase names for *function-level* ones.

problem because there would be no definite type to replace 'A with in the signature of its `eval` method – so we would not know how to type expressions such as `x.eval` when `x` has type `EvalBase`. Note that 'A can even be instantiated to *several incomparable types* within a single class, if `EvalBase` is inherited several times.

What is interesting here is that MLscript infers a **this** type refinement (also called *self type*), which specifies what the type of **this** should be for the mixin to be well-typed. Here, **this** represents the final object obtained from the future mixin composition. Crucially, notice that the type of `eval` is *no longer recursive* – indeed, it no longer contains a recursive ‘**as**’ binder. This is because we have *opened* the recursion, and the type that is inferred for `eval` *precisely* specifies what this partial definition accomplishes: it examines the top level of an expression and when that expression is an `Add`, it calls `eval` open-recursively through **this** with the corresponding subexpressions, expecting integer results from that recursive call.

Opening recursion in this way allows us to adapt the interpretation of this partially-specified recursive function, as we shall see shortly.

Closing back. We can immediately tie the knot and obtain an equivalent implementation to the original recursive function `eval` by defining a class that only inherits from `EvalBase`:

```
class SimpleLang extends EvalBase
```

whose inferred type signature is:

```
class SimpleLang: {
  eval: (Lit | Add['a] as 'a) → Int
}
```

Something important happened here: by creating the class `SimpleLang` from the previous mixin, we effectively *tie the recursive knot* for the corresponding method. That is, to type check `SimpleLang`, MLscript constrains the “open” polymorphic type variable 'A associated with `eval` in `EvalBase` and instantiates it to the correct type to make the overall mixin composition type check. More specifically, remember that `eval` as defined in `EvalBase` was given type $(\text{Lit} \mid \text{Add}['A]) \rightarrow \text{Int}$ *assuming* that **this** had type $\{ \text{eval}: ('A) \rightarrow \text{Int} \}$. Here, we know that the type of **this** is `SimpleLang` and that `SimpleLang`'s `eval` implementation is the one inherited from `EvalBase`. So when constraining types to make the subtyping relation $\text{SimpleLang} <: \{ \text{eval}: ('A) \rightarrow \text{Int} \}$ hold, this leads to constraining $(\text{Lit} \mid \text{Add}['A]) \rightarrow \text{Int} <: ('A) \rightarrow \text{Int}$, which in turn leads to the constraint $'A <: (\text{Lit} \mid \text{Add}['A])$. So MLscript instantiates the type variable 'A to the principal solution, i.e the recursive type $(\text{Lit} \mid \text{Add}['a]) \text{ as } 'a$, which satisfies this recursive constraint.

Extending the operations. Now consider extending our code for a new expression pretty-printing method:

```
mixin PrettyBase {
  fun print(e) = if e is
    Lit(n)          then toString(n)
    Add(lhs, rhs)  then this.print(lhs) ++ "+" ++ this.print(rhs) }
```

Mixin `PrettyBase` defines a `print` method for `Lit` and `Add`. Its inferred type is analogous to that of `EvalBase`. This demonstrates that we can extend the operations performed on our simple language, which is one of the extensibility directions considered by the Expression Problem.

11:8 super-Charging Object-Oriented Programming

Extending the data types. Next, consider another direction of code extension – defining a *new expression constructor*. We here define a negation expression type `Neg`:

```
class Neg[T](expr: T)
```

Now, the obvious question is how to extend arbitrary existing operations to this new data type constructor in a way that is as general and modular as possible.

super-charging OOP with Polymorphic Mixins

As noticed by Garrigue [16], it is often useful to define components that extend *yet unknown* base implementations, so that the same components can be applied to different base implementations, and so that in general we can merge independently-defined languages together. This is possible to do in MLscript by defining mixins that make use of **this** and **super**, as in the following example:

```
mixin EvalNeg {  
  fun eval(e) =  
    if e is Neg(d) then 0 - this.eval(d)  
    else super.eval(e)  
}
```

which can be written more concisely using the following syntax sugar:

```
mixin EvalNeg { fun eval(override Neg(d)) = 0 - this.eval(d) }
```

We can include this partial `Neg`-handling recursion step as part of any previously-defined base implementation, such as our previous `EvalBase`. We get the following inferred type for `EvalNeg`, which precisely describes this property:

```
mixin EvalNeg:  $\forall$  'A 'B 'R . {  
  this: { eval: 'A  $\rightarrow$  Int }  
  super: { eval: 'B  $\rightarrow$  'R }  
  eval: (Neg['A] | 'B \ Neg)  $\rightarrow$  (Int | 'R)  
}
```

We can see that the type signature of our mixin now includes a **super** refinement *in addition* to the **this** refinement. This is the key to enabling polymorphic extension: when composing such a mixin later on, MLscript will match up this **super** requirement with whatever implementation is provided by the previous mixin implementations in the chain of mixin composition. Recursive knots will only be tied when the mixin is composed as part of a class.

The `PrettyNeg` extension for pretty-printing is defined analogously.

Tying the knot again. Finally, we can compose everything together as part of a new class:

```
class Lang extends EvalBase, EvalNeg, PrettyBase, PrettyNeg
```

And here is the type signature inferred for this definition:

```
class Lang: {  
  eval: (Lit | Add['a] | Neg['a] as 'a)  $\rightarrow$  Int  
  print: (Lit | Add['a] | Neg['a] as 'a)  $\rightarrow$  Str  
}
```

Again, what happens here is important to consider. We are now tying the knot with respect to *both* **this** and **super** in all the mixins making up the mixin inheritance stack. More specifically, we start by making sure that the member types provided by the first mixin `EvalBase` satisfy the **super** requirement of the second mixin `EvalNeg`, then we compute new member types based on `EvalNeg`'s contributions, before checking that the resulting type

satisfies the **super** requirement of the next mixin in line, `PrettyBase`, etc. This results in the inferred recursive types above, which precisely characterize what shapes of data that `Lang`'s `eval` and `print` methods can handle.

Polymorphic extensibility. To demonstrate that our `EvalNeg` component is truly generic over the existing implementation it is to be merged upon, we can define yet another mixin that adds a new `Mul` language feature:

```
class Mul[T](lhs: T, rhs: T)
mixin EvalMul { fun eval(override Mul(l, r)) = this.eval(l) * this.eval(r) }
```

And then we compose all of these mixins together in two possible orders (the order determines which of `Neg` and `Mul` will be matched first):

```
class LangNegMul extends EvalBase, EvalNeg, EvalMul
class LangMulNeg extends EvalBase, EvalMul, EvalNeg
```

In both cases, the inferred signature is equivalent:

```
class LangNegMul: { eval: (Lit | Add['a'] | Neg['a'] | Mul['a'] as 'a) → Int }
```

Pattern-Matching All the Way

To conclude this motivating example, we exemplify a capability of our system that most solutions to the expression problem lack, with the notable exception of polymorphic variants (see Section 4.3): the ability of *pattern matching deeply inside subexpressions*, which enables the definition of optimization passes.

For instance, below we define an `EvalNegNeg` optimization which shortcuts the evaluation of double negations, directly evaluating the doubly-negated expression instead:

```
mixin EvalNegNeg { fun eval(override Neg(Neg(d))) = this.eval(d) }
```

of inferred type:

```
mixin EvalNegNeg: ∀ 'A 'B 'C 'D . {
  super: {eval: (Neg['A'] | 'B) → 'C}
  this: {eval: 'D → 'C}
  fun eval: (Neg[Neg['D']] | 'A \ Neg | 'B \ Neg) → 'C
}
```

This type deserves some explanation. The parameter type of `eval` is `'Neg[Neg['D']] | 'A \ Neg | 'B \ Neg`, which describes the fact that:

- `eval` accepts either an instance of `Neg` or, failing that, a `'B` that is *not* a `Neg`;
- If the argument *is* a `Neg`, then its type argument must itself be either a `Neg` or an `'A` that is not a `Neg`;
- If that nested type argument is a `Neg`, then its type argument must be `'D`. Since this type argument is passed to `this.eval`, we get the **this** refinement `{eval: 'D → 'C}`.
- In case either the `eval` argument is not a `Neg` (so the argument is a `'B`) or the `eval` argument is a `Neg['A']` where `'A` is not a `Neg`, evaluation falls back to a **super** call, which is translated into the **super** refinement `{eval: (Neg['A'] | 'B) → 'C}`.

This mixin can be merged onto any mixin stack to obtain the desired effect; for example:¹⁰

```
class Lang extends EvalBase, EvalNeg, EvalMul, EvalNegNeg
```

¹⁰In this case, it is important to mix in `EvalNegNeg` *after* `EvalNeg` in the inheritance stack, so that the optimization semantics override the base semantics, and not the other way around. This is a fundamental property of optimization passes: their composition order matters.

3 A Core Language for SuperOOP

In this section, we present an explicitly-typed core language that captures the core object-oriented concepts of SuperOOP, leaving type inference aside. We first informally present the key innovation of SuperOOP's object-oriented type system and then define λ^{super} , a minimal declarative and explicitly-polymorphic calculus.

3.1 SuperOOP Core Concepts

The core concepts of SuperOOP can be summarized as follows.

Interfaces, mixins, and classes. Interfaces, mixins, and classes are three orthogonal building blocks that model OOP in our system. *Interfaces* define a set of method signatures. For an object conforming to an interface, it should support all the methods specified in that interface. Contrary to classes and mixins, which in our core language have no types, we associate each interface with its own type. *Mixins* provide *implementations* for methods. *Classes*, finally, implement interfaces by a set of parameters, which represent the *state* of the object, and a linear composition of mixins.

Interface inheritance. As in most OOP languages, existing interfaces can be extended with additional methods through interface inheritance. A child interface may inherit from several parent interfaces (i.e., we support *multiple inheritance* of interfaces). Moreover, a child interface may override parent method signatures with *refined* signatures, as determined by the subtyping relation. As an example, consider the following interface composition:

```
interface I1 { a: S }; interface I2 { a: T }; interface I3 extends I1, I2
```

Method `a`'s signature in the composed interface `I3` is the *intersection* of the inherited signatures, i.e. `S & T`. Intersection types enable precise multiple interface inheritance, since they are used as greatest lower bounds of the inherited type signatures, which also makes the composed interface a subtype of all inherited interfaces.

Mixin composition. SuperOOP mixins are compositional and reusable building blocks to construct classes. They provide partial method implementations that, when composed together, are checked to satisfy the interface that the class is meant to conform to. A mixin composition is simply a list of mixins. Each mixin in a mixin composition *overrides* not only method implementations but also method *types* inherited from previous mixins. So the type of a method may change along the mixin composition, but the type system ensures that the typing assumptions made by each implementation (in the form of `this` and `super` refinements) are satisfied. This also explains why mixins are not considered types (unlike, e.g., Scala traits): the fact that a mixin is present in the inheritance clause of a class does *not* imply that the resulting object will offer methods with types comparable to the ones provided by the mixin.

Precisely-Typed Open Recursion. A crucial feature of OOP, *open recursion* is the ability for a method to invoke itself or another method via a late-bound `this` instance, which may lead to evaluating overriding implementations. In most OOP languages with inheritance, the type of `this` is the current class's type. In these languages, method invocations on `this` are safe because overriding implementations from subclasses can only refine the types of overridden methods. By contrast, in SuperOOP, methods are overridden regardless of types,

Names, types, and terms	
<i>Class name</i>	C
<i>Mixin name</i>	M, N
<i>Interface name</i>	I, J
<i>Field name</i>	m, p
<i>Type</i>	$S, T, U, V ::= X, Y \mid I[\overline{T}] \mid S \rightarrow T \mid \forall X. T \mid S \& T \mid \mathbf{Object}$
<i>Term</i>	$e ::= x, y \mid \mathbf{this} \mid \mathbf{super} \mid \lambda x : T. e \mid \Lambda X. e$ $\mid e_1 e_2 \mid e T \mid e.m \mid \mathbf{new} C[\overline{T}](\overline{e})$
Interfaces, mixins, and classes	
<i>Structural type</i>	$\mathcal{R} ::= \{ \overline{m : \overline{T}} \}$
<i>Implementation</i>	$\mathcal{I} ::= \{ \overline{m : \overline{T} = e} \}$
<i>Top-level definition</i>	$\mathcal{D} ::=$
<i>Interface</i>	$I[\overline{X}] \triangleleft J[\overline{T}] \mathcal{R}$
<i>Mixin</i>	$\mid M[\overline{X}]_T^{\mathcal{R}} \mathcal{I}$
<i>Class</i>	$\mid C[\overline{X}](\overline{m : \overline{T}}) \triangleleft I[\overline{T}], \overline{M[\overline{T}]}$
<i>Program</i>	$\mathcal{P} ::= \overline{\mathcal{D}}; e$

■ **Figure 1** Syntax of $\lambda^{\mathbf{super}}$.

and the actual type of **this** is only decided when the mixin composition is finalized as part of a class definition. Therefore, a precise type specification for **this** is necessary for open recursive calls in mixin methods. Importantly, **this** type refinement can be polymorphic at the mixin level, being instantiated at mixin composition time (i.e., upon being used as part of a class definition). Such polymorphism allows for later extensions to the shapes of data types that a method may be made to work on, as described in Section 2.

3.2 Formal Syntax

We now introduce the $\lambda^{\mathbf{super}}$ calculus, a formalization of SuperOOP. The design of this calculus is inspired by Featherweight Generic Java [19] and Pathless Scala [20]. Throughout our formalization, we use the notation $\overline{E}_i^{i \in n..m}$ to denote the repetition of syntax form E_i with index i from n to m . We use \overline{E} as a shorthand when i is not necessary for disambiguation. Moreover, we use $[T/X]$ to denote the conventional capture-avoiding substitution of a list of type parameters \overline{X} (which can possibly be empty) to \overline{T} . In definitions of metafunctions, we use \emptyset as a default vacuous result.

The syntax of $\lambda^{\mathbf{super}}$ is presented in Figure 1. Meta-variables S, T, U, V range over types, which include type variables, interfaces with a list of type arguments, arrow types, universally quantified types, intersection types, and the top type **Object**. For terms e , there are term variables x and y . **this** and **super** are akin to term variables with special treatment. We have standard explicitly-typed lambda abstractions and term applications, as well as type abstraction and type application terms. Method invocation and access to object fields share a single syntax: we consider access to object fields as method invocation. Objects are created with a **new** keyword with term and type arguments supplied.

$\boxed{S <: T}$	S-REFL $\frac{}{T <: T}$	S-TOP $\frac{}{T <: \text{Object}}$	S-INTERFACE $\frac{S \in \text{parents}(I[\overline{T}])}{I[\overline{T}] <: S}$	S-INV $\frac{S <: T \quad T <: S}{I[S] <: I[\overline{T}]}$
S-ANDL $\frac{}{S_1 \ \& \ S_2 <: S_1}$	S-ANDR $\frac{}{S_1 \ \& \ S_2 <: S_2}$	S-AND $\frac{S <: T_1 \quad S <: T_2}{S <: T_1 \ \& \ T_2}$	S-TRANS $\frac{S <: U \quad U <: T}{S <: T}$	
	S-ARROW $\frac{S_2 <: S_1 \quad T_1 <: T_2}{S_1 \rightarrow T_1 <: S_2 \rightarrow T_2}$	S-FORALL $\frac{S <: T}{\forall X. S <: \forall X. T}$		

■ **Figure 2** Declarative subtyping.

The top-level definitions of λ^{super} are interfaces, mixins, and classes. Every interface $I[\overline{X}]$ has a type parameter list $[\overline{X}]$, a structural refinement \mathcal{R} , and inherits multiple parent interfaces $J[\overline{T}]$. A structural refinement \mathcal{R} contains a list of method signatures $m : T$ that specify methods' names and types. Mixins, parametrized by type parameters, provide method implementations \mathcal{I} . Crucially, each mixin has a structural refinement \mathcal{R} attached to **super** and a type T for **this** for precise typing of open recursion. Finally, a class has a class-level type parameter list, immutable object fields, an interface it implements, and a mixin composition $M[\overline{T}]$ that provides method implementations. A program consists in a list of top-level definitions and a term that accesses them. For all top-level definitions, we require the standard well-formedness conditions that all names are uniquely defined and no class transitively inherits itself. In later rules, we assume terms' access to the underlying top-level definitions.

3.3 Static Semantics

We present the static semantics of λ^{super} which includes a declarative subtyping, term typing, and well-formedness check of top-level definitions.

Declarative subtyping. Figure 2 shows the declarative subtyping of λ^{super} . Most rules are unsurprising. Rule S-INTERFACE describes that an interface is a subtype of its parent interfaces. Auxiliary function $\text{parents}(I[\overline{T}])$ (defined in the extended version) returns the list of parent interfaces. For simplicity, we consider that interfaces are *invariant* in their type parameters (rule S-INV). A universally quantified type is a subtype of another universally quantified type only when they are quantifying the same type variable.

Term typing. Figure 3 lists the typing rule of terms. $\Gamma \vdash e : T$ is the term typing relation. A typing context Γ maps term variables to types, **super** to a structural refinement, and **this** to a type. The typing rules for term variables (T-VAR), lambda and type abstractions (T-ABS and T-TABS), term and type applications (T-APP and T-TAPP), as well as the subsumption rule (T-SUB), are standard. Note that since **super** is not bound to a type (but to a structural refinement) in typing contexts, **super** itself will never be assigned a type, which matches the usual semantics of **super** that it should only receive method call messages but not be passed around. The typing of method invocations is separated into two cases. If the receiver is a term (other than **super**) that has a type, we look up the method signature

$$\begin{array}{c}
\text{Typing context} \quad \Gamma ::= \epsilon \mid \Gamma, x : T \mid \Gamma, \mathbf{super} : \mathcal{R} \mid \Gamma, \mathbf{this} : T \\
\\
\boxed{\Gamma \vdash e : T} \quad \begin{array}{c} \text{T-VAR} \\ \Gamma(x) = T \\ \hline \Gamma \vdash x : T \end{array} \quad \begin{array}{c} \text{T-THIS} \\ \Gamma(\mathbf{this}) = T \\ \hline \Gamma \vdash \mathbf{this} : T \end{array} \quad \begin{array}{c} \text{T-ABS} \\ \Gamma, x : S \vdash e : T \\ \hline \Gamma \vdash \lambda x : S. e : S \rightarrow T \end{array} \quad \begin{array}{c} \text{T-TABS} \\ \Gamma \vdash e : T \\ \hline \Gamma \vdash \Lambda X. e : \forall X. T \end{array} \\
\\
\begin{array}{c} \text{T-APP} \\ \Gamma \vdash e_1 : S \rightarrow T \quad \Gamma \vdash e_2 : S \\ \hline \Gamma \vdash e_1 e_2 : T \end{array} \quad \begin{array}{c} \text{T-TAPP} \\ \Gamma \vdash e : \forall X. S \\ \hline \Gamma \vdash e T : [T/X]S \end{array} \quad \begin{array}{c} \text{T-ACCESS} \\ \Gamma \vdash e : T \quad \text{mtype}(m, T) = S \\ \hline \Gamma \vdash e.m : S \end{array} \\
\\
\begin{array}{c} \text{T-SUPER} \\ \Gamma(\mathbf{super}) = \mathcal{R} \\ \text{mrefn}(m, \mathcal{R}) = S \\ \hline \Gamma \vdash \mathbf{super}.m : S \end{array} \quad \begin{array}{c} \text{T-NEW} \\ \text{vparams}(C[\overline{T}]) = \overline{m_i : U_i^{i \in 1..n}} \\ \Gamma \vdash e_i : \overline{U_i^{i \in 1..n}} \quad \text{ctype}(C[\overline{T}]) = V \\ \hline \Gamma \vdash \mathbf{new } C[\overline{T}](\overline{e_i^{i \in 1..n}}) : V \end{array} \quad \begin{array}{c} \text{T-SUB} \\ \Gamma \vdash e : S \\ S <: T \\ \hline \Gamma \vdash e : T \end{array}
\end{array}$$

Given that interface I is defined as $I[\overline{X}] \triangleleft J[\overline{U}] \mathcal{R}$:

$$\begin{array}{l}
\text{mtype}(m, I[\overline{T}]) = \begin{cases} [\overline{T/X}]S & \text{if } (m : S) \in \mathcal{R} \\ S & \text{if } m \notin \mathcal{R} \text{ and } \text{mtype}(m, \&[\overline{T/X}]J[\overline{U}]) = S \end{cases} \\
\text{mtype}(m, S \& T) = \begin{cases} U \& V & \text{if } \text{mtype}(m, S) = U \text{ and } \text{mtype}(m, T) = V \\ U & \text{if } \text{mtype}(m, S) = U \text{ and } \text{mtype}(m, T) = \emptyset \\ V & \text{if } \text{mtype}(m, S) = \emptyset \text{ and } \text{mtype}(m, T) = V \end{cases} \\
\text{mtype}(m, T) = \emptyset \text{ otherwise}
\end{array}$$

■ **Figure 3** Term typing.

in the receiver's type. Function $\text{mtype}(m, T)$ computes method m 's signature from type T . Otherwise, if the receiver is **super**, we directly read the method type from its associated structural refinement using function $\text{mrefn}(m, \mathcal{R})$ (defined in the extended version). To type class instantiation (T-NEW), we check that all constructor arguments match the types of the class fields returned by function $\text{vparams}(C[\overline{T}])$, and the object has interface type $\text{ctype}(C[\overline{T}])$ of the class (vparams and ctype are defined in the extended version).

The design of mtype basically follows that of Pathless Scala [20]. When a method signature is present in an interface, we directly return it. Otherwise, we search parent interfaces by calling mtype with the *intersection* of all parent interfaces (denoted as $\&J[\overline{U}]$). Note that nullary intersection is **Object**. To compute a method signature from an intersection type, we recursively consider both sides of the intersection. When both types define the method, we take the intersection of corresponding results.

Well-formedness of top-level definitions. Figure 4 shows the well-formedness check of mixins, classes, and interfaces. We put name lookup results of those structures as premises in the rules. The first premises of rules in Figure 4 are the case.

11:14 super-Charging Object-Oriented Programming

$$\begin{array}{c}
\boxed{M \text{ ok}} \quad \frac{\text{MIXINCHECK} \quad M[\overline{X}]_T^{\mathcal{R}} \{ \overline{m : S = e} \} \quad \forall (m : S = e) \in M. \text{ this} : T, \text{ super} : \mathcal{R} \vdash e : S}{M \text{ ok}} \\
\\
\boxed{I \text{ ok}} \quad \frac{\text{INTERFACECHECK} \quad I[\overline{X}] \triangleleft J[\overline{T}] \{ \overline{m : S} \} \quad \overline{J \text{ ok}} \quad \forall (m : S) \in I. \text{ mtype}(m, \&\mathcal{J}[\overline{T}]) = \emptyset \text{ or } \begin{cases} \text{mtype}(m, \&\mathcal{J}[\overline{T}]) = U \\ S <: U \end{cases}}{I \text{ ok}} \\
\\
\boxed{C \text{ ok}} \quad \frac{\text{CLASSCHECK} \quad C[\overline{X}](\overline{p : T}) \triangleleft I[\overline{U}], \overline{M_i[\overline{U}]}^{i \in n..1} \quad \begin{array}{l} I \text{ ok} \quad \overline{M_i \text{ ok}} \quad \overline{M_i} \Rightarrow \overline{C} \quad \forall m \in \text{mnames}(I[\overline{U}]). \begin{cases} \text{mtype}(m, I[\overline{U}]) = S \\ \text{search}(m, 0, C) = V \\ V <: S \end{cases} \end{array}}{C \text{ ok}} \\
\\
\boxed{M_i \Rightarrow C} \quad \frac{\text{INHERITCHECK} \quad C[\overline{X}](\overline{p : U'}) \triangleleft I[\overline{U}], \overline{M_i[\overline{V}]}^{i \in n..1} \quad M_i[\overline{Y}]_T^{\mathcal{R}} \mathcal{I} \quad I[\overline{U}] <: [\overline{V/Y}]T \quad \forall (m : S) \in \mathcal{R}. \begin{cases} \text{search}(m, (i+1), C) = S' \\ S' <: [\overline{V/Y}]S \end{cases}}{M_i \Rightarrow C}
\end{array}$$

■ **Figure 4** Well-formedness check of top-level definitions and mixin inheritance check.

Well-formed mixins. To check a mixin ($M \text{ ok}$), we check that every method implementation can be typed at its signature with precise types of **this** and **super** in the context.¹¹

Well-formed interfaces. An interface is well-formed ($I \text{ ok}$) when its parent interfaces are all well-formed. A method signature should either be newly introduced (in this case, $\text{mtype}(m, \&\mathcal{J}[\overline{T}]) = \emptyset$), or have a subtype of the intersection of all m 's signatures in parents (i.e., $\text{mtype}(m, \&\mathcal{J}[\overline{T}]) = U$).

Well-formed classes. Class well-formedness check ($C \text{ ok}$) considers the following aspects:

1. The implemented interface and each mixin in the mixin composition are well-formed.
2. Open-recursive calls via **this** in the mixin composition are safe: the class type is a subtype of each mixin's **this** type annotation.
3. The mixin composition is correct: each mixin's structural refinement on **super** is satisfied.
4. The interface is satisfied: the class has all methods (and fields, as we uniformly treat fields and methods) required, and their signatures conform to the interface.

¹¹ Note that we bind **this** to a type while **super** to a structural refinement in each mixin. For **super**, the parent mixin in the composition hierarchy does *not* define an object type. It is therefore enough to give **super** a structural method refinement to tell what types the overridden methods should have. On the other hand, **this** is late-bound to the receiver object that has a type, can be passed around, and receive method invocation messages. Hence **this** is annotated with a type, and the annotated type should be a supertype of the later defined class's type.

Given that class C is defined as $C[\overline{X}](\overline{m_j : T_j}) \triangleleft I[\overline{S}], \overline{M_i[\overline{S}]}^{i \in n..1}$,
and mixin M_i is defined as $M_i[\overline{Y}]_V^R \mathcal{I}$:

$$\begin{aligned} \text{search}(m_j, 0, C) &= \begin{cases} T_j & \text{if } m_j : T_j \in \overline{m_j : T_j} \\ U & \text{if } m_j \notin \overline{m_j : T_j} \text{ and } \text{search}(m_j, 1, C) = U \end{cases} \\ \text{search}(m, i, C) &= \begin{cases} [\overline{S/\overline{Y}}]U & \text{if } 0 < i \leq n \text{ and } (m : U = e) \in \mathcal{I} \\ U & \text{if } 0 < i \leq n \text{ and } m \notin \mathcal{I} \text{ and } \text{search}(m, (i + 1), C) = U \end{cases} \\ \text{search}(m, i, C) &= \emptyset \text{ otherwise} \end{aligned}$$

■ **Figure 5** Method implementation type search function.

For 1., I **ok** checks the interface, and \overline{M} **ok** checks each mixin. Relation $M_i \Rightarrow C$ implements mixin inheritance check which deals with 2. and 3.. It checks if the inheritance of the i -th mixin in class C 's mixin composition is correct. Note that the index i here ranges in $n..1$ (as $\overline{M_i[\overline{S}]}^{i \in n..1}$), which means syntactically, the *rightmost* mixin in the mixin composition is the *first* one. Rule INHERITCHECK guarantees that, first, **this** type of the i -th mixin should be a *supertype* of the interface that the class conforms to, which satisfies 2.. Second, for each method m 's signature in the structural refinement of **super**, the parent mixin composition provides a compatible implementation. Specifically, the type of m 's implementation provided by mixins ranging in $n..(i + 1)$ (computed by $\text{search}(m, (i + 1), C)$, defined in Figure 5 and explained later) should be a *subtype* of the i -th mixin's **super** refinement on m , which satisfies 3.. To satisfy 4., for each method name m defined in the interface (computed by mnames , defined in the extended version), its implementation type provided by the class fields or mixin composition (computed by $\text{search}(m, 0, C)$) should be compatible with the signature specified by the interface (computed by mtype).

Method implementation type search. Figure 5 defines function $\text{search}(m, i, C)$ to search implementation type of m provided by fields or mixins ranging in $n..i$. When $i = 0$, it searches class fields for the method name m . If m is not implemented by fields, the search continues with the first mixin ($i = 1$). For the i -th mixin, the search directly returns the method signature if m is implemented in this mixin. Otherwise, it continues with the parent mixin (indexed $(i + 1)$). The search returns \emptyset if i exceeds the length of class C 's mixin composition ($i > n$), which means m is not implemented in the class, and the search fails.

3.4 Dynamic Semantics

Figure 6 lists the syntax of values, results, and runtime contexts, and lists the evaluation rules that produce values (the rules that produce runtime errors are omitted and can be found in the extended version). The big-step evaluation judgment $\Xi \vdash e \Downarrow r$ denotes that term e evaluates to result r under runtime context Ξ . The result of evaluation may be a normal value or an error. Values are either *closures* or *objects*. A runtime context Ξ binds values to term variables and a *configured object* to **this**. A configured object $\{i \star C[\overline{T}](\overline{v})\}$ is a pair of an object and a natural number i called the *search index*. This index directs the search for method implementation in the object fields and mixin composition at runtime. The evaluation rules for variables and term applications are standard. For type applications, while we use type substitution in the semantics, this will be no-op at runtime, as all generic types are erasable – only class tags are used at runtime, which are concrete types that need

11:16 super-Charging Object-Oriented Programming

<i>Value</i>	$v, w ::= \langle \lambda x : T. e, \Xi \rangle \mid \langle \Lambda X. e, \Xi \rangle \mid C[\overline{T}](\overline{v})$		
<i>Runtime context</i>	$\Xi ::= \epsilon \mid \Xi, x \mapsto v \mid \Xi, \mathbf{this} \mapsto \{i \star C[\overline{T}](\overline{v})\}$		
<i>Result</i>	$r ::= \mathbf{val} v \mid \mathbf{err}$		
$\Xi \vdash e \Downarrow r$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50%;"> $\frac{\text{E-VAR} \quad \Xi(x) = v}{\Xi \vdash x \Downarrow \mathbf{val} v}$ </td> <td style="text-align: center; width: 50%;"> $\frac{\text{E-THIS} \quad \Xi(\mathbf{this}) = \{i \star C[\overline{T}](\overline{v})\}}{\Xi \vdash \mathbf{this} \Downarrow \mathbf{val} C[\overline{T}](\overline{v})}$ </td> </tr> </table>	$\frac{\text{E-VAR} \quad \Xi(x) = v}{\Xi \vdash x \Downarrow \mathbf{val} v}$	$\frac{\text{E-THIS} \quad \Xi(\mathbf{this}) = \{i \star C[\overline{T}](\overline{v})\}}{\Xi \vdash \mathbf{this} \Downarrow \mathbf{val} C[\overline{T}](\overline{v})}$
$\frac{\text{E-VAR} \quad \Xi(x) = v}{\Xi \vdash x \Downarrow \mathbf{val} v}$	$\frac{\text{E-THIS} \quad \Xi(\mathbf{this}) = \{i \star C[\overline{T}](\overline{v})\}}{\Xi \vdash \mathbf{this} \Downarrow \mathbf{val} C[\overline{T}](\overline{v})}$		
	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50%;"> $\frac{\text{E-APP} \quad \begin{array}{l} \Xi \vdash e_1 \Downarrow \mathbf{val} \langle \lambda x : T. e, \Xi' \rangle \\ \Xi \vdash e_2 \Downarrow \mathbf{val} v \quad \Xi', x \mapsto v \vdash e \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash e_1 e_2 \Downarrow \mathbf{val} v'}$ </td> <td style="text-align: center; width: 50%;"> $\frac{\text{E-TAPP} \quad \begin{array}{l} \Xi \vdash e \Downarrow \mathbf{val} \langle \Lambda X. e', \Xi' \rangle \\ [T/X]\Xi' \vdash [T/X]e' \Downarrow \mathbf{val} v \end{array}}{\Xi \vdash e T \Downarrow \mathbf{val} v}$ </td> </tr> </table>	$\frac{\text{E-APP} \quad \begin{array}{l} \Xi \vdash e_1 \Downarrow \mathbf{val} \langle \lambda x : T. e, \Xi' \rangle \\ \Xi \vdash e_2 \Downarrow \mathbf{val} v \quad \Xi', x \mapsto v \vdash e \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash e_1 e_2 \Downarrow \mathbf{val} v'}$	$\frac{\text{E-TAPP} \quad \begin{array}{l} \Xi \vdash e \Downarrow \mathbf{val} \langle \Lambda X. e', \Xi' \rangle \\ [T/X]\Xi' \vdash [T/X]e' \Downarrow \mathbf{val} v \end{array}}{\Xi \vdash e T \Downarrow \mathbf{val} v}$
$\frac{\text{E-APP} \quad \begin{array}{l} \Xi \vdash e_1 \Downarrow \mathbf{val} \langle \lambda x : T. e, \Xi' \rangle \\ \Xi \vdash e_2 \Downarrow \mathbf{val} v \quad \Xi', x \mapsto v \vdash e \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash e_1 e_2 \Downarrow \mathbf{val} v'}$	$\frac{\text{E-TAPP} \quad \begin{array}{l} \Xi \vdash e \Downarrow \mathbf{val} \langle \Lambda X. e', \Xi' \rangle \\ [T/X]\Xi' \vdash [T/X]e' \Downarrow \mathbf{val} v \end{array}}{\Xi \vdash e T \Downarrow \mathbf{val} v}$		
	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50%;"> $\frac{\text{E-ABS}}{\Xi \vdash \lambda x : T. e \Downarrow \mathbf{val} \langle \lambda x : T. e, \Xi \rangle}$ </td> <td style="text-align: center; width: 50%;"> $\frac{\text{E-TABS}}{\Xi \vdash \Lambda X. e \Downarrow \mathbf{val} \langle \Lambda X. e, \Xi \rangle}$ </td> </tr> </table>	$\frac{\text{E-ABS}}{\Xi \vdash \lambda x : T. e \Downarrow \mathbf{val} \langle \lambda x : T. e, \Xi \rangle}$	$\frac{\text{E-TABS}}{\Xi \vdash \Lambda X. e \Downarrow \mathbf{val} \langle \Lambda X. e, \Xi \rangle}$
$\frac{\text{E-ABS}}{\Xi \vdash \lambda x : T. e \Downarrow \mathbf{val} \langle \lambda x : T. e, \Xi \rangle}$	$\frac{\text{E-TABS}}{\Xi \vdash \Lambda X. e \Downarrow \mathbf{val} \langle \Lambda X. e, \Xi \rangle}$		
	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50%;"> $\frac{\text{E-NEW} \quad \begin{array}{l} \text{vparams}(C[\overline{T}]) = \overline{m}_i \\ \Xi \vdash e_i \Downarrow \mathbf{val} v_i \end{array}}{\Xi \vdash \mathbf{new} C[\overline{T}](\overline{e}_i) \Downarrow \mathbf{val} C[\overline{T}](\overline{v}_i)}$ </td> <td style="text-align: center; width: 50%;"> $\frac{\text{E-ACCESS} \quad \begin{array}{l} \Xi \vdash e \Downarrow \mathbf{val} C[\overline{S}](\overline{v}) \\ (\mathbf{this} \mapsto \{0 \star C[\overline{S}](\overline{v})\}) \vdash \mathbf{super}.m \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash e.m \Downarrow \mathbf{val} v'}$ </td> </tr> </table>	$\frac{\text{E-NEW} \quad \begin{array}{l} \text{vparams}(C[\overline{T}]) = \overline{m}_i \\ \Xi \vdash e_i \Downarrow \mathbf{val} v_i \end{array}}{\Xi \vdash \mathbf{new} C[\overline{T}](\overline{e}_i) \Downarrow \mathbf{val} C[\overline{T}](\overline{v}_i)}$	$\frac{\text{E-ACCESS} \quad \begin{array}{l} \Xi \vdash e \Downarrow \mathbf{val} C[\overline{S}](\overline{v}) \\ (\mathbf{this} \mapsto \{0 \star C[\overline{S}](\overline{v})\}) \vdash \mathbf{super}.m \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash e.m \Downarrow \mathbf{val} v'}$
$\frac{\text{E-NEW} \quad \begin{array}{l} \text{vparams}(C[\overline{T}]) = \overline{m}_i \\ \Xi \vdash e_i \Downarrow \mathbf{val} v_i \end{array}}{\Xi \vdash \mathbf{new} C[\overline{T}](\overline{e}_i) \Downarrow \mathbf{val} C[\overline{T}](\overline{v}_i)}$	$\frac{\text{E-ACCESS} \quad \begin{array}{l} \Xi \vdash e \Downarrow \mathbf{val} C[\overline{S}](\overline{v}) \\ (\mathbf{this} \mapsto \{0 \star C[\overline{S}](\overline{v})\}) \vdash \mathbf{super}.m \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash e.m \Downarrow \mathbf{val} v'}$		
	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; width: 50%;"> $\frac{\text{E-ARGMISS} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{0 \star C[\overline{S}](\overline{v})\} \quad m \notin \text{vparams}(C[\overline{S}]) \\ (\mathbf{this} \mapsto \{1 \star C[\overline{S}](\overline{v})\}) \vdash \mathbf{super}.m \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash \mathbf{super}.m \Downarrow \mathbf{val} v'}$ </td> <td style="text-align: center; width: 50%;"> $\frac{\text{E-ARGHIT} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{0 \star C[\overline{S}](\overline{v}_i)\} \\ \text{vparams}(C[\overline{S}]) = \overline{m}_i : \overline{U}_i \end{array}}{\Xi \vdash \mathbf{super}.m_i \Downarrow \mathbf{val} v_i}$ </td> </tr> </table>	$\frac{\text{E-ARGMISS} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{0 \star C[\overline{S}](\overline{v})\} \quad m \notin \text{vparams}(C[\overline{S}]) \\ (\mathbf{this} \mapsto \{1 \star C[\overline{S}](\overline{v})\}) \vdash \mathbf{super}.m \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash \mathbf{super}.m \Downarrow \mathbf{val} v'}$	$\frac{\text{E-ARGHIT} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{0 \star C[\overline{S}](\overline{v}_i)\} \\ \text{vparams}(C[\overline{S}]) = \overline{m}_i : \overline{U}_i \end{array}}{\Xi \vdash \mathbf{super}.m_i \Downarrow \mathbf{val} v_i}$
$\frac{\text{E-ARGMISS} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{0 \star C[\overline{S}](\overline{v})\} \quad m \notin \text{vparams}(C[\overline{S}]) \\ (\mathbf{this} \mapsto \{1 \star C[\overline{S}](\overline{v})\}) \vdash \mathbf{super}.m \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash \mathbf{super}.m \Downarrow \mathbf{val} v'}$	$\frac{\text{E-ARGHIT} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{0 \star C[\overline{S}](\overline{v}_i)\} \\ \text{vparams}(C[\overline{S}]) = \overline{m}_i : \overline{U}_i \end{array}}{\Xi \vdash \mathbf{super}.m_i \Downarrow \mathbf{val} v_i}$		
	$\frac{\text{E-SUPERMISS} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \quad i > 0 \\ m \notin \text{methods}(i, C[\overline{S}](\overline{v})) \quad (\mathbf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash \mathbf{super}.m \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash \mathbf{super}.m \Downarrow \mathbf{val} v'}$		
	$\frac{\text{E-SUPERHIT} \quad \begin{array}{l} \Xi(\mathbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \quad i > 0 \\ (m : U = e) \in \text{methods}(i, C[\overline{S}](\overline{v})) \quad (\mathbf{this} \mapsto \{(i+1) \star C[\overline{S}](\overline{v})\}) \vdash e \Downarrow \mathbf{val} v' \end{array}}{\Xi \vdash \mathbf{super}.m \Downarrow \mathbf{val} v'}$		

■ **Figure 6** Big-step operational semantics producing values.

no substitution. Note that evaluation of **this** is to simply read the configured object from the context and return a plain object (i.e., with no search index). Class instantiations produce objects. Lambda and type abstractions are evaluated to closures. Note that $\lambda^{\mathbf{super}}$ would not need a value restriction [32] even if we added imperative effects to it, because it does not evaluate under polymorphic abstractions. This is different from the real MLscript language, which does need a value restriction as it uses ML-style polymorphism.

Method invocation and access to fields. Proper modeling of method invocation and access to fields are of our particular interest. The following procedure explains the overall idea:

1. When the receiver is a term (modulo **super**), we first evaluate the term to an object and search through the object's fields for the method implementation (E-ACCESS).
2. If the invoking method is not provided by any object field, we traverse the mixin composition of the class (E-ARGMISS).

$$\begin{array}{c}
\boxed{v : T} \\
\hline
\text{VT-ABS1} \\
\Gamma \models \Xi \quad \Xi(\mathbf{this}) = \{i \star C[\overline{U}](\overline{v})\} \\
\mathcal{R} \models \{i \star C[\overline{U}](\overline{v})\} \\
\Gamma, x : S, \mathbf{super} : \mathcal{R} \vdash e : T \\
\hline
\langle \lambda x : S. e, \Xi \rangle : S \rightarrow T
\end{array}
\quad
\begin{array}{c}
\text{VT-TABS1} \\
\Gamma \models \Xi \quad \Xi(\mathbf{this}) = \{i \star C[\overline{S}](\overline{v})\} \\
\mathcal{R} \models \{i \star C[\overline{S}](\overline{v})\} \\
\Gamma, \mathbf{super} : \mathcal{R} \vdash e : T \\
\hline
\langle \Lambda X. e, \Xi \rangle : \forall X. T
\end{array}
\quad
\begin{array}{c}
\text{VT-SUB} \\
v : S \\
S <: T \\
\hline
v : T
\end{array}$$

$$\boxed{\mathcal{R} \models \{i \star C[\overline{S}](\overline{v})\}} \quad \frac{C[\overline{X}](\overline{n} : \overline{T}') \triangleleft I[\dots], \overline{M}[\dots] \quad \forall (m : T) \in \mathcal{R}. \begin{cases} \text{search}(m, i, C) = U \\ [\overline{S}/\overline{X}]U <: T \end{cases}}{\mathcal{R} \models \{i \star C[\overline{S}](\overline{v})\}}$$

■ **Figure 7** Value typing of closures.

3. If the invoking method is provided as an object field, we return the value bound to the field (E-ARGHIT).
4. If the invoking method is not implemented by the i -th mixin, we search the next mixin in the composition hierarchy (E-SUPERMISS). Helper function $\text{methods}(i, C[\overline{S}](\overline{v}))$ (defined in the extended version) returns all method implementations of the i -th mixin.
5. If the invoking method is implemented by the i -th mixin, we evaluate the method body with \mathbf{this} bound to the configured object where the search index points to the parent mixin (E-SUPERHIT).

3.5 Metatheory

We now develop the metatheory of $\lambda^{\mathbf{super}}$. We follow Ernst et al.'s approach to prove type soundness of our big-step style semantics.

Value typing. Our metatheory focuses on *strong* soundness, that is, we need to type values to ensure that the evaluation result keeps the type. Value typing rules of closures are listed in Figure 7. Rule VT-ABS1 types lambda abstraction body under a typing context Γ with the term variable bound to the input type and \mathbf{super} refined by a structural refinement \mathcal{R} . Here we perform two consistency checks. First, the typing context should be consistent with the runtime context ($\Gamma \models \Xi$, rules are listed in the extended version), i.e., each term variable is bound to a value that matches the variable's type in the typing context. Second, to guarantee that calls to super implementations are always safe, the structural refinement \mathcal{R} giving precise types to calls on \mathbf{super} in the closure body should be *consistent* with the configured object in the closure's context. Relation $\mathcal{R} \models \{i \star C[\overline{S}](\overline{v})\}$ implements the second consistency check, which examines each method signature's compatibility with the method implementation type provided by the configured object. The remaining rules (in the extended version) that type objects and closures with no binding to \mathbf{this} in the context are non-surprising.

Soundness. We finally show the soundness results of our formal calculus. The complete proofs can be found in the extended version. For a program \mathcal{P} , we denote its top-level definitions as $\overline{\mathcal{D}}_{\mathcal{P}}$ and the associated term as $e_{\mathcal{P}}$. The preservation lemma is stated below:

► **Lemma 1** (Preservation). *If $\overline{\mathcal{D}}_{\mathcal{P}} \text{ ok}$ and $\epsilon \vdash e_{\mathcal{P}} : T$ and $\epsilon \vdash e_{\mathcal{P}} \Downarrow r$ then $r = \mathbf{val} \ v$ and $v : T$.*

We define the *finite evaluation* relation [9] here to augment our big-step semantics with fuel.

11:18 super-Charging Object-Oriented Programming

► **Definition 2** (Finite evaluation). Define an evaluation relation $\Xi \vdash e \Downarrow_k r^+$ (where $r^+ ::= r \mid \text{kill}$, and k is the step-counting index, i.e. fuel) with evaluation rules copied from $\Xi \vdash e \Downarrow r$. For each rule, \Downarrow in the conclusion is replaced by \Downarrow_k , and \Downarrow in premises is replaced by \Downarrow_{k-1} . Also, propagate timeout result of subderivations (the corresponding rules are listed in the extended version). Finally, add the following axiom:

$$\begin{array}{l} E\text{-TIMEOUT} \\ \Xi \vdash e \Downarrow_0 \text{kill} \end{array}$$

The soundness theorem of our calculus follows from the preservation lemma that rules out errors when evaluation terminates and the coverage lemma that ensures our evaluation rules with finite fuel always produce a result.

► **Lemma 3** (Coverage). For all n, Ξ , and e , there exists an r^+ such that $\Xi \vdash e \Downarrow_n r^+$.

► **Definition 4** (Expression divergence). e diverges \triangleq For all $n, \epsilon \vdash e \Downarrow_n \text{kill}$.

► **Theorem 5** (Soundness). If $\overline{\mathcal{D}_{\mathcal{P}} \text{ok}}$ and $\epsilon \vdash e_{\mathcal{P}} : T$ then (1) $\epsilon \vdash e_{\mathcal{P}} \Downarrow \text{val } v$ and $v : T$, or (2) $e_{\mathcal{P}}$ diverges.

4 Discussion and Related Work

We now discuss the expressiveness, limitations, and implementation of SuperOOP as presented in this paper, and we compare our approach to related work.

4.1 Expressiveness and Limitations

Thanks to the clear separation of concerns between the orthogonal concepts of interfaces, mixins, and classes, and thanks to the flexibility of mixins, SuperOOP not only captures standard OOP features but can also be used to explain existing advanced OOP models.

Desugaring traditional classes. A classic OOP class is desugared into three SuperOOP core language components: (a) a core-language class for its fields; (b) a core-language mixin for its implementations; and (c) a core-language interface for its method signatures. Although our core language does not directly support class inheritance, this feature can easily be desugared into SuperOOP. For example, recall `ColoredPoint` from Section 1, which inherited from class `Point`. This class hierarchy can be desugared to SuperOOP as:

```
interface IPoint { x: Int; y: Int }
class Point(x: Int, y: Int) implements IPoint

interface IColoredPoint extends IPoint, Colored
class ColoredPoint(x: Int, y: Int, color: Color) implements IColoredPoint
```

Multiple inheritance and linearization. Languages that support multiple inheritance usually have a *linearization* mechanism that determines the order of inherited parent classes, traits, or mixins. The underlying assumption is that each parent can only be inherited at most once, so if a parent transitively occurs more than once in an inheritance clause, the linearization mechanism removes all but its first occurrence. Consequently, linearization affects the semantics of method resolution and **super**-calls. For example, Scala uses linearization for its multiple trait inheritance system [22]. The linearization of a Scala class definition of the form `class C extends B0, B1, ..., Bn` starts with `B0`'s linearization and appends to it the

linearization of B_1 save for those traits that are already in the constructed linearization of B_0 , etc. Several languages such as Python adopt the influential C3 linearization algorithm [1]. Although SuperOOP does not natively support multiple class inheritance, we can still apply any linearization algorithms used by existing languages and desugar the result using core SuperOOP classes, interfaces, and mixins. On the other hand, in SuperOOP, one can inherit a given mixin an arbitrary number of times at different positions in the mixin inheritance stack. The resolution of method invocations simply follows the order of inherited mixins, which do not necessarily need to be linearized. So SuperOOP’s approach is more general.

We show an example encoding of Scala multiple trait inheritance in SuperOOP in the extended version of this paper.

Mixin parameters. Mixin parameters are a powerful extension to the core SuperOOP language presented in this paper. They for instance allow one to define flexible and efficient streaming processing abstractions that are composed through mixins, as in the following:

```
module MyPipeline extends
  Map(x => x + 1),
  Filter(x => x % 2 == 0),
  Map(x => x * 2)
```

We use *two* instances of `Map` in the mixin composition above, showing that using **this** refinements to encode mixin parameters would not be sufficient, as each of these two `Map` instances needs to be given a *different* argument. Mixin parameters are implemented in MLscript/SuperOOP, but we omitted this extension from λ^{super} for simplicity.

Member access control. We have not yet modeled in the core language nor implemented any notions of encapsulation and visibility, such as the **private** and **protected** modifiers. We expect that modeling these features should be straightforward, as their design is mostly orthogonal to the features of SuperOOP.

4.2 Implementation of SuperOOP in MLscript

We now briefly describe our implementation and possible alternative implementation strategies.

Compilation to JavaScript. MLscript currently compiles to JavaScript, which supports classes as first-class entities. This means it is possible to define mixins directly, by using functions. For instance, the `EvalNeg` and `EvalMul` mixins and the `LangNegMul` class mentioned in Section 2 are essentially compiled into the following JavaScript code:

```
function mkEvalNeg(base) {
  return class EvalNeg extends base {
    eval(e) {
      if (e instanceof Neg) return 0 - this.eval(e.expr)
      else return super.eval(e) } }
}
function mkEvalMul(base) {
  return class EvalMul extends base {
    eval(e) {
      if (e instanceof Mul) return this.eval(e.lhs) * this.eval(e.rhs)
      else return super.eval(e) } }
}
class LangNegMul extends mkEvalMul(mkEvalNeg(EvalBase))
```

11:20 super-Charging Object-Oriented Programming

One side effect of this straightforward implementation is that mixins in MLscript can be inherited an arbitrary number of times and that no inheritance linearization is needed. MLscript *classes*, on the other hand, follow the usual single-inheritance hierarchy discipline, which is useful for type checking pattern matching and inferring simple types for it.

Compilation to other targets. We are also considering adding alternative compilation backends to MLscript, such as backend compilers targeting WebAssembly and the Java Virtual Machine. In that context, we can still follow the general JavaScript-based semantics described above, but we will make sure to evaluate the mixin functions at compilation time, to guarantee optimal performance and simple compilation. Super calls would then be resolved statically, allowing for efficient target code. Therefore, our approach to mixin composition should offer better performance than alternative solutions to the expression problem which rely on closure compositions and thus require virtual dispatch, like the approach of Garrigue [16]. However, we reserve a rigorous performance evaluation for future work.

Separate compilation. An aspect of the Expression Problem as originally stated is that it should be possible to compile each extension separately before putting them all together. We can essentially achieve this even in the static compiler scenario by separately compiling method *implementations* and composing classes whose methods simply forward to these pre-compiled implementations. This is more or less the approach used by Scala for traits, which was shown to be practical in real-world scenarios.

Case studies. In the extended version of this paper, we provide case studies of MLscript/SuperOOP that include a modular evaluator of extended lambda calculus, as described by Garrigue [16], and a simple “regions” DSL developed by Sun et al. [31]. These case studies showcase the flexibility of SuperOOP polymorphic mixins, the ability to handle mutually-recursive functions across different mixins, interpret complex data types, and optimize domain-specific languages via built-in nested pattern matching. Additionally, thanks to MLscript’s powerful principal type inference [26], those case studies type check without the help of a single type annotation.

4.3 Solutions to the Expression Problem

There is a sea of work in extensible programming that address the Expression Problem, based on techniques such as polymorphic variants [15] in OCaml, recursive modules [21] in ML, and new programming paradigms [4, 23] like Compositional Programming [34]. We survey a few of them by showing their solutions to the Expression Problem and discuss various design tradeoffs with respect to the approach of SuperOOP.

Polymorphic Variants. The *polymorphic variant* (PV) solution [16] probably comes closest to our approach. Open recursion there is implemented by way of an explicit parameter for recursive calls, and by manually tying the recursive knots. For example, one defines an open-recursive base implementation of evaluation on two expression data types as follows:

```
let eval_base eval_rec = function
  | 'Lit(n) → n
  | 'Add(e1, e2) → eval_rec e1 + eval_rec e2
(* val eval_base :
  ('a → int) → [< 'Add of 'a * 'a | 'Lit of int ] → int *)
```

PVs differ from traditional variants or *algebraic data types* (ADTs) in that PVs allow the use of arbitrary constructors without a corresponding data type definition; they can be thought of as ADTs that are “not fully specified” and thus allow further extension. In the example above, two constructors ‘Lit and ‘Add are introduced. Function `eval_base` takes a first parameter `eval_rec` for open-recursive calls and the expression to evaluate as a second parameter. Parameter `eval_rec` accepts expressions with type ‘a, and the expression is required to have type [`< ‘Add of 'a * 'a | ‘Lit of int`], which allows either an ‘Add expression containing nested subexpressions of type ‘a, or a ‘Lit instance with an integer payload. Extending this base evaluator with new operations is done by composing it inside new functions. To extend the supported expression forms, one defines another evaluation implementation that works, e.g., on negations:

```
let eval_ext eval_rec = function
  'Neg(e) → 0 - eval_rec e
(* val eval_ext : ('a → int) → [< ‘Neg of 'a ] → int *)
```

Finally, one needs to tie both implementations together:

```
type 'a expr_base = ['Lit of int | ‘Add of 'a * 'a]
type 'a expr_ext = ['Neg of 'a]
let rec eval = function
  | #expr_base as x → eval_base eval x
  | #expr_ext as x → eval_ext eval x
(* val eval :
  ([< ‘Add of 'a * 'a | ‘Lit of int | ‘Neg of 'a ] as 'a) → int *)
```

Function `eval` dispatches the evaluation of the base and extended data types to the two evaluation sub-implementations, and it ties the recursive knots by passing itself as the entry point of the recursion. Note that `eval` has an inferred recursive type that accepts an expression recursively constructed by the three variants. Compared with our solution, from a programming style perspective, one programs with polymorphic variants in a functional way, while SuperOOP adopts a more object-oriented style. More importantly, polymorphic variants suffer from several practical drawbacks, including loss of polymorphism and approximated typing of pattern matching [5]. Those drawbacks can be fixed by embracing “proper” implicit subtyping as in MLscript [26]. In particular, we argue that union types are simpler than row polymorphism, which imperfectly emulates subtyping through unification [26].

OCaml’s Object System. In OCaml class definitions, one can annotate “self” with a type signature and define “super” explicitly in a way that superficially looks similar to SuperOOP. One may be tempted to try and encode precise typing of open recursion in OCaml, to enable extensible programming with classes. However, this does not work due to OCaml’s use of unification and its lack of subtyping: the self and super types are *unified* with the object type being defined, and thus all three must exactly coincide. By contrast, SuperOOP mixins allows *different* self and super types and allows overriding methods with *different* types, which is crucial for our technique. We discuss this in more detail in the extended version.

Featherweight Generic Go. Go is a popular programming language developed by Google. Featherweight Go (FG) and its generic version Featherweight Generic Go (FGG) proposed by Griesemer et al. [18] are formal developments of Go with the goal of helping “get polymorphism right”. FGG provides a solution to the Expression Problem based on generics and covariant matching of method receiver type refinements, as in:

```
func (e Plus(type a Evaluator)) Eval() int {
  return e.left.Eval() + e.right.Eval()
}
```

11:22 super-Charging Object-Oriented Programming

Method `Eval` is generic in type variable `'a'` which is upper-bounded by interface `Evaler`. Once dissociated from the quantification of `a`, the receiver type of the method is `Plus(a)`, the type of a `Plus` instance with subexpressions of type `'a'`. To extend the supported operations in the encoded language, one may define a similar pretty-printing method. Finally, one combines the interfaces for different interpretations together in a final expression type:

```
type Expr interface {
    Evaler
    Stringer
}
```

Type `Expr` composes two operations together, so it implements both of `Evaler` and `Stringer` (an interface for stringification). One can build and use such expressions as follows:

```
var e Expr = Plus(Expr){Lit{1}, Lit{2}}
var result Int = e.Eval()
var pretty string = e.String()
```

While this allows FGg to solve the Expression Problem, the features that enable this solution are not part of the Go team's current design for generics [18]. Moreover, the inspection of data structures only happens at the *outermost* level. If one wants to deeply transform an expression instance, that is, to inspect its inner structure and, for example, perform optimizations on it, one would have to make an additional method to delegate the inspection semantics itself. This approach, called *delegated method patterns* in Sun et al.'s work [31], is non-modular in FGg as it requires adding a new method for each inner structure inspection and to implement this method for each constructor of the data type, even those constructors that should otherwise fall into a default case of the encoded pattern matching.

Object Algebras. *Object Algebras* are a well-known object-oriented approach to solve the Expression Problem [23]. The key to this solution is an abstract factory called *object algebra interface*, which contains data type constructor signatures, leaving their interpretation unspecified. An object algebra interface for expressions could be, in Scala syntax:

```
trait ExpAlg[Exp] {
    def Lit: Int => Exp
    def Add: (Exp, Exp) => Exp
}
```

Trait `ExpAlg` specifies two data type constructors, and it is parameterized by type parameter `Exp` that indicates the interpretation of expression data types. We can implement evaluation on expressions by implementing the object algebra interface:

```
trait IEval { def eval: Int }
trait Eval extends ExpAlg[IEval] {
    def Lit = n => new IEval { def eval = n }
    def Add = (e1, e2) => new IEval { def eval = e1.eval + e2.eval }
}
```

Trait `Eval` is an object algebra which implements `ExpAlg` with the type parameter instantiated to `IEval`. Trait `IEval` indicates that expressions can be evaluated to integers. To extend the language with new operations, we may simply define a new interpretation type and a corresponding object algebra interface implementation. On the other hand, for new data type extensions, we inherit the object algebra interface and the old implementation:

```

trait NegAlg[Exp] extends ExpAlg[Exp] {
  def Neg: Exp => Exp
}
trait EvalNeg extends NegAlg[IEval] with Eval {
  def Neg = (e) => new IEval { def eval = 0 - e.eval }
}

```

We can now define an expression instance and instantiate the language:

```

trait exp[Exp](f: NegAlg[Exp]) {
  f.Add(f.Lit(1), f.Neg(f.Lit(-1)))
}
object eval extends EvalNeg
println(exp(eval).eval)

```

In trait `exp`, the data type constructors are accessed through the input object algebra `f`. With different implementations of the object algebra interface passed in, the expression will be interpreted in different ways. However, as noticed by Zhang et al. [34], one needs to create an expression instance for each data type interpretation, and there is no built-in approach to composing interpretations in different object algebras. Moreover, as data type constructors are specified through type *signatures* in object algebra interfaces, there is no way to have an inspectable representation of language instances without a complete definition of abstract syntax, blocking useful extensions such as modular transformations and optimizations.

Compositional Programming. *Compositional programming* [34] (implemented in the *CP* language) is a novel programming paradigm that features modularity. It supports a *merge operator* as the introduction term for intersection types. At the type level, the intersection type operator composes interfaces. At the term level, the merge operator composes *first-class traits* that contain data and operations. Similarly to Object Algebras, in Compositional Programming, a *compositional interface* specifies data type signatures, leaving their interpretation unspecified, and concrete interpretations are defined in first-class traits:

```

type ExpSig<Exp> = {
  Lit : Int → Exp;
  Add : Exp → Exp → Exp;
};
type Eval = { eval : Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};

```

Trait `evalNum` implements the compositional interface `ExpSig<Eval>` which specifies that `Lit` and `Add` support an evaluation method. Similarly, one can implement a pretty-printing operation by adding another concrete interpretation. To extend the expression language with new data types, one extends the compositional interface and implements new operations in derived traits. Finally, everything is tied together with the merge operator as shown below:

```

type NegSig<Exp> extends ExpSig<Exp> = {
  Neg : Exp → Exp → Exp;
};
evalNeg = trait implements NegSig<Eval> inherits evalNum => {
  (Neg e).eval = 0 - e.eval;
};

```

```

exp Exp = trait [self : NegSig<Exp>] => {
  test = new Neg (new Add (new Lit 1) (new Lit 2));
};
// Assume pretty-printing of expression is analogously defined
e = new evalNeg ,, printNeg ,, exp @(Eval & Print);

```

Trait `exp` contains an example expression. The `self` type annotation in square brackets enables the trait body to access the three data type constructors. With the merge operator, trait instance `e` is composed of traits that contain different expression interpretations and the test trait. Note that trait `Exp` is passed with an intersection type argument `Eval & Print`, meaning the expression language supports both evaluation and pretty-printing.

In recent follow-up work on Compositional Programming by Sun et al. [31], different aspects of domain-specific language embedding are investigated, including the two-direction extensibility of language constructs and their interpretations, transformations and optimizations on language instances, etc. Since Compositional Programming does not natively support nested pattern matching (unlike our approach), deep inspection of data is only possible via the delegated method pattern (discussed above in the paragraph on `Go`), which is “not as convenient”, as the authors put it. We also argue that this does *not* work well for defining *optimizations* in a modular way. Indeed, optimizations are fundamentally order-sensitive, and encoding them in terms of CP’s unordered patterns requires non-local transformations of the involved pattern matching structures. For instance, one cannot *independently* define optimizations for evaluating `Neg(Neg(e))` as `e` and `Neg(Lit(n))` as `Lit(0 - n)`, whereas doing so in `MLscript/SuperOOP` is straightforward.

Approaches lacking type safety. It is much easier to solve the Expression Problem if one no longer cares about catching composition errors at compilation time. Zenger and Odersky [33] propose to use exception-throwing default cases in base implementations and to override these cases in further extensions, which relies on the programmer *remembering* to override all default cases and to pass only supported expression forms to the various methods in the program. Similar to `SuperOOP`, in a method that defines the interpretation of extended data types and overrides the base interpretation, they delegate the interpretation of base data types to the overridden method using `super`. While just as flexible as `SuperOOP`, this approach is fundamentally unsafe and error-prone. Going further, at the other end of the spectrum, approaches such as monkey-patching and Julia-style multiple dispatch allow completely dynamic updates of base implementations, which trivially supports extension but is anti-modular, as reasoning about the well-foundedness of method calls on given argument types requires global knowledge of all extension points in the program and libraries.

4.4 Modeling Inheritance and Reuse

In this subsection, we discuss previous work related to modeling inheritance and code reuse.

In their seminal *Inheritance Is Not Subtyping* paper, Cook et al. [7] introduced the crucial idea that inheritance could be unrestrained if it was decoupled from the subtyping relationship. However, they do not provide a specific source language in which to realize their ideas and only describe an imagined typed encoding of it, without an obvious way of connecting that encoding back to a hypothetical source language.

Bracha and Cook [3] describe both a Smalltalk-style approach and a CLOS-style multiple inheritance approach for modeling single inheritance and `super`. The paper uses a notion of implementation “deltas” Δ , which are not first-class and only used for explanation. In our approach, this notion of deltas exists as a first-class entity which we call *mixins*. Bracha and

Cook describe mixins as a form of *abstraction* (over an unknown base class), and linearization as *application* (wiring in all the base classes), by analogy with the classical lambda calculus concepts. In our approach, abstraction is similarly done through **super** and application is done through **extends**, but we do not require linearization and allow mixins to be inherited an arbitrary number of times. While Bracha and Cook leverage the notion that subtyping is not inheritance and allow the types of methods to change, they do not support the idea of precise **this** and **super** annotations and thus cannot precisely type open recursion.

The concept of “mixin” described by Flatt et al. [13, 14, 12] is related to ours, but conceptually different. While they do model **super**, their mixins necessarily conform to interfaces and are thus constrained to specific method signatures, preventing SuperOOP-style modular programming. The authors discuss the possibility of solving the EP with modules and their mixins in later work [11], but without proposing a static typing model.

Schärli et al. [29] study and discuss many perceived problems with mixin composition. They suggest that *traits* are a better unit of abstraction. We agree that traits are useful for architecting OOP code in the large, but argue that mixins are independently useful: abstract (i.e., open-ended) base classes are specifically what unlocks the expressiveness of mixin inheritance and our new solution to the Expression Problem. We believe that mixins should be conceptualized as pure *whitebox implementation* bundles (the implementation itself being the API) by contrast with interfaces, which hide implementation detail, and traits, which enable a form of well-behaved (associative and commutative) multiple inheritance, and that all three could have a place in an OO programmer’s toolkit.

The idea of separating reusable components from types was previously embraced by Bettini et al. [2], who argue that the role of *units of reuse* and the role of *types* are competing, as also observed by Cook et al. [7] and Snyder [30]. The semantics of Bettini et al.’s trait systems are similar to Schärli et al.’s but provide additional flexibility, in that traits are composed with explicit operations on methods such as renaming and exclusion to resolve conflict. A similar idea is used by Damiani et al. [8] in their design of a language enabling both trait reuse and *deltas* of classes, in the context of Software Product Line Engineering.

Type classes as in languages like Haskell [27] and Scala [24] also provide *data abstraction* and powerful parametrization and extensibility [6]. SuperOOP’s **super** is a way of *nesting* interpretations the same way one can design dependent type class instances. Any class hierarchy encoded solely with **super** refinements in SuperOOP translates straightforwardly to classic type classes. However, type classes *per se* are not enough for modular code reuse with recursive data structures, as that requires open recursion. Explicit encodings of open recursion can be implemented in Haskell and Scala, but these would live outside of the type class definitions and are orthogonal to type classes. By contrast, SuperOOP directly provides precisely-typed open recursion via **this** refinements in mixins.

5 Conclusion and Future Work

We presented a new approach to OOP which cleanly separates the concerns of *state*, *implementations*, and *interfaces* into the orthogonal constructs of *classes*, *mixins*, and *interfaces*. We showed that a refined typing of mixins allows for a new and powerful solution to the expression problem. Finally, we presented an implementation in MLscript, leveraging its flexible type inference capabilities to enable annotation-free modular programming. The main item of future work we would like to look into is the *deep* composition of mixin *families*, reminiscent of Delta-Oriented Programming [28, 8] but with precisely-typed open recursion.

References

- 1 Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 69–82, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/236337.236343.
- 2 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Stocco. Traitrecordj: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013. Special section: Principles and Practice of Programming in Java 2009/2010 & Special section: Self-Organizing Coordination. doi:10.1016/j.scico.2011.06.007.
- 3 Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/97945.97982.
- 4 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, September 2009. doi:10.1017/S0956796809007205.
- 5 Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 378–391, Nara, Japan, September 2016. Association for Computing Machinery. doi:10.1145/2951913.2951928.
- 6 William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1640089.1640133.
- 7 William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 125–135, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/96709.96721.
- 8 Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. A unified and formal programming model for deltas and traits. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, pages 424–441, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
- 9 Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1111037.1111062.
- 10 Andong Fan. Simple extensible programming through precisely-typed open recursion. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2022, pages 54–56, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3563768.3563951.
- 11 Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 94–104, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/289423.289432.
- 12 Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In Naoki Kobayashi, editor, *Programming Languages and Systems*, pages 270–289, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- 13 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 171–183, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/268946.268961.


- 14 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. *A Programmer's Reduction Semantics for Classes and Mixins*, pages 241–269. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. doi:10.1007/3-540-48737-9_7.
- 15 Jacques Garrigue. Programming with polymorphic variants. In *In ACM Workshop on ML*, 1998. URL: https://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf.
- 16 Jacques Garrigue. Code reuse through polymorphic variants. In *In Workshop on Foundations of Software Engineering*, 2000. URL: <https://www.math.nagoya-u.ac.jp/~garrigue/papers/variant-reuse.pdf>.
- 17 David S. Goldberg, Robert Bruce Findler, and Matthew Flatt. Super and inner: Together at last! In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 116–129, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1028976.1028987.
- 18 Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. Featherweight go. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428217.
- 19 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. doi:10.1145/503502.503505.
- 20 Guillaume Martres. Pathless scala: A calculus for the rest of scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala*, SCALA 2021, pages 12–21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3486610.3486894.
- 21 Keiko Nakata and Jacques Garrigue. Recursive modules for programming. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 74–86, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1159803.1159813.
- 22 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language, 2004.
- 23 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 2–27, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_2.
- 24 Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1869459.1869489.
- 25 Lionel Parreaux. The ultimate conditional syntax. *ML Family Workshop*, 2022. URL: <https://icfp22.sigplan.org/details/mlfamilyworkshop-2022-papers/6/The-Ultimate-Conditional-Syntax>.
- 26 Lionel Parreaux and Chun Yin Chau. MLstruct: Principal type inference in a boolean algebra of structural types. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563304.
- 27 Simon Peyton Jones. Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13, January 2003.
- 28 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 77–91, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 29 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, pages 248–274, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

- 30 Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, pages 38–45, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/28697.28702.
- 31 Yaozhu Sun, Utkarsh Dhandhanian, and Bruno C. d. S. Oliveira. Compositional embeddings of domain-specific languages. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563294.
- 32 Andrew K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4):343–355, December 1995. doi:10.1007/BF01018828.
- 33 Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 241–252, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/507635.507665.
- 34 Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. Compositional programming. *ACM Trans. Program. Lang. Syst.*, 43(3), September 2021. doi:10.1145/3460228.

LoRe: A Programming Model for Verifiably Safe Local-First Software (Extended Abstract)

Julian Haas  

Technische Universität Darmstadt, Germany

Ragnar Mogk 

Technische Universität Darmstadt, Germany

Elena Yanakieva 

University of Kaiserslautern-Landau, Germany

Annette Bieniusa 

University of Kaiserslautern-Landau, Germany

Mira Mezini 

Technische Universität Darmstadt, Germany

Abstract

Local-first software manages and processes private data locally while still enabling collaboration between multiple parties connected via partially unreliable networks. Such software typically involves interactions with users and the execution environment (the outside world). The unpredictability of such interactions paired with their decentralized nature make reasoning about the correctness of local-first software a challenging endeavor. Yet, existing solutions to develop local-first software do not provide support for automated safety guarantees and instead expect developers to reason about concurrent interactions in an environment with unreliable network conditions.

We propose *LoRe*, a programming model and compiler that automatically verifies developer-supplied safety properties for local-first applications. *LoRe* combines the declarative data flow of reactive programming with static analysis and verification techniques to precisely determine concurrent interactions that violate safety invariants and to selectively employ strong consistency through coordination where required. We propose a formalized proof principle and demonstrate how to automate the process in a prototype implementation that outputs verified executable code. Our evaluation shows that *LoRe* simplifies the development of safe local-first software when compared to state-of-the-art approaches and that verification times are acceptable.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Software and its engineering → Distributed programming languages; Software and its engineering → Data flow languages; Software and its engineering → Consistency; Theory of computation → Pre- and post-conditions; Theory of computation → Program specifications; Computer systems organization → Peer-to-peer architectures

Keywords and phrases Local-First Software, Reactive Programming, Invariants, Consistency, Automated Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.12

Related Version *Extended Version*: <https://arxiv.org/abs/2304.07133> [13]

Supplementary Material *Software (Source Code)*: <https://github.com/stg-tud/LoRe>
Software (ECOOP 2023 Artifact Evaluation approved artifact):
<https://doi.org/10.4230/DARTS.9.2.11>

Funding This work was funded by the German Federal Ministry of Education and Research together with the Hessen State Ministry for Higher Education (ATHENE), the German Research Foundation (SFB 1053), and the German Federal Ministry for Economic Affairs and Climate Action project SafeFBDC (01MK21002K).



© Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 12; pp. 12:1–12:15



Leibniz International Proceedings in Informatics
LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Applications that enable multiple parties connected via partially unreliable networks to collaboratively process data prevail today. An illustrative example is a distributed calendar application with services to add or modify appointments, where a user may maintain multiple calendars on different devices, may share calendars with other users, back them up in a cloud; calendars must be accessible to users in a variety of scenarios, including offline periods, e.g., while traveling – yet, planning appointments may require coordination between multiple parties. The calendar application is representative for other collaborative data-driven software such as group collaboration tools, digital (cross-organizational) supply chains, multiplayer online gaming, and more.

The dominating software architecture for such applications is centralized: data is collected, managed, and processed centrally in data centers, while devices on the edge of the communication infrastructure serve primarily as interfaces to users and the outside world. This architecture simplifies the software running on edge devices since concerns like consistent data changes to ensure safety properties are managed centrally. However, this comes with issues including loss of control over data ownership and privacy, insufficient offline availability, poor latency, inefficient use of communication infrastructure, and waste of (powerful) computing resources on the edge.

To address these issues, local-first principles for software development have been formulated [17], calling for moving data management and processing to edge devices instead of confining the data to clouds. But for programming approaches that implement these principles to be viable alternatives to the centralized approach, they must support automatically verifiable safety guarantees to counter for the simplifying assumptions afforded by a centralized approach. Unfortunately, existing approaches to programming local-first applications such as *Yjs* [31] or *Automerge*¹ do not provide such guarantees. They use *conflict-free replicated data types (CRDTs)* [34] to store the parts of their state that is shared across devices and rely on callbacks for modeling and managing state that changes in both time and space. The unpredictability of the interactions triggered by the outside world, concurrently at different devices, paired with the absence of a central authority and the prevailing implicit dependencies in current callback-centred programming models, makes such reasoning without automated support a challenging, error-prone endeavour.

To close this gap, we propose a programming model for local-first applications that features explicit safety properties and automatically enforces them. The model has three core building blocks: *reactives*, *invariants*, and *interactions*. *Reactives* express values that change in time, but also in space by being replicated over multiple devices. *Invariants* are formula in first-order logic specifying safety properties that must hold at all times when the application interacts with the outside world, or values of reactives are observable. *Interactions* interface to the outside world and encapsulate changes to all reactives affected by interactions with it (state directly changed by the interactions, device-local values derived from the changed state, and shared state at remote devices). We use automatic verification with invariants as verification obligations to identify interactions that need coordination across devices, for which the compiler generates the coordination protocol; all other interactions become visible in causal order. This way, the compiler makes an application-specific availability-safety trade-off.

¹ <https://automerge.org/>

In summary, we make the following contributions²:

1. A programming model for local-first applications with verified safety properties (Section 2), called LoRe. While individual elements of the model, e.g., CRDTs or reactivities, are not novel, they are repurposed, combined, and extended in a unique way to systematically address specific needs of local-first applications with regard to ensuring safety properties.
2. A formal definition of the model including a formal notion of invariant preservation and confluence for interactions, and a modular verification that invariants are never violated. In particular, our model enables invariants that reason about the sequential behaviour of the program. In case of potential invariant violation due to concurrent execution, LoRe automatically adds the necessary coordination logic (see the extended version of this work²).
3. A verifying compiler³ that translates LoRe programs to Viper [28] for automated verification and to Scala for the application logic including synthesized synchronization to guarantee the specified safety invariants (Section 3).
4. An evaluation of LoRe in two case studies (Section 4). Our evaluation validates two claims we make about the programming model proposed, (a) It facilitates the development of safe local-first software, and (b) it enables an efficient and modular verification of safety properties. It further shows that the additional safety properties offered by our model do not come with prohibitive costs in terms of verification effort and time.

2 LoRe in a Nutshell

We introduce the concepts of LoRe along the example of a distributed calendar for tracking work meetings and vacation days. LoRe is an external DSL that compiles to Scala (for execution) and Viper IR [28] (for verification); its syntax is inspired by both. A LoRe program defines a distributed application that runs on multiple physical or virtual devices.⁴ Listing 1 shows a simplified implementation of the calendar example application in LoRe. As any LoRe program, it consists of replicated state (`Source` reactivities in Lines 2-3), local values derived from them (`Derived` reactivities in Lines 5-6), interactions (Lines 8-15), and invariants (Lines 20-23).

2.1 Reactives

Reactivities are the composition units in a LoRe program. We distinguish two types of them: *source* and *derived* reactivities, declared by the keywords `Source` and `Derived`, respectively. Source reactivities are values that are directly changed through interactions. Their state is modeled as *conflict-free replicated data types* (CRDTs) [34, 32] and is replicated between the different devices collaborating on the application. Derived reactivities represent local values that are automatically computed by the system from the values of other reactivities (source or derived). Changes to source reactivities automatically (a) trigger updates of derived reactivities and (b) cause devices to asynchronously send update messages to the other devices, which then merge the changes into their local state. Together, local propagations and asynchronous

² This is a short version of this work. The extended version is available at: <https://doi.org/10.48550/arXiv.2304.07133>.

³ The source code of our prototype implementation is available at <https://github.com/stg-tud/LoRe>.

⁴ We assume that every device is running the same application code (i.e., the same binary), and different types of devices (such as client and server) are modeled by limiting them to execute a subset of the defined interactions.

■ **Listing 1** The distributed calendar application.

```

1  type Calendar = ASet[Appointment]
2  val work: Source[Calendar] = Source(ASet())
3  val vacation: Source[Calendar] = Source(ASet())
4
5  val all_appointments: Derived[Set[Appointment]] = Derived{ work.
      toSet.union(vacation.toSet) }
6  val remaining_vacation: Derived[Int] = Derived{ 30 - sumDays(vacation.
      toSet) }
7
8  val add_appointment : Unit = Interaction[Calendar][Appointment]
9      .requires{ cal => a => get_start(a) < get_end(a) }
10     .requires{ cal => a => !(a in cal.toSet)}
11     .executes{ cal => a => cal.add(a) }
12     .ensures { cal => a => a in cal.toSet }
13  val add_vacation : Unit = add_appointment.modifies(vacation)
14     .requires{ cal => a => remaining_vacation - a.days >= 0}
15  val add_work      : Unit = add_appointment.modifies(work)
16
17  UI.display(all_appointments, remaining_vacation)
18  UI.vacationDialog.onConfirm{a => add_vacation.apply(a)}
19
20  invariant forall a: Appointment ::
21      a in all_appointments ==> get_start(a) < get_end(a)
22
23  invariant remaining_vacation >= 0

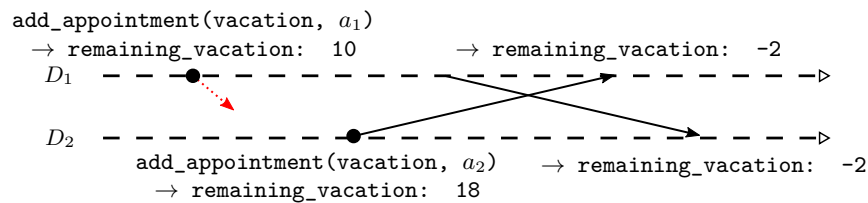
```

cross-device update messages ensure that users always have a consistent view of the overall application state. All reactivities are statically declared in the program source code. LoRe then statically extracts knowledge about the data flow for modular verification and to minimize the proof goals (cf. Sec 3.1). We discuss the technical implications of static reactivities in Section 6.

Listing 1 shows two source reactivities, `work` and `vacation` (Line 2 and 3), each modeling a calendar as a set of appointments. The work calendar tracks work meetings, while the vacation calendar contains registered vacation days. When defining a source reactive, programmers have to choose a CRDT for the reactive’s internal state. LoRe offers a selection of pre-defined CRDTs including various standard data types such as sets, counters, registers and lists. Further data types can be supported by providing a Viper specification for that data type. In this case, an *add-wins-set* (a set CRDT where additions have precedence over concurrent deletions) is selected for both source reactivities. Appointments from both calendars are tracked in the `all_appointments` derived reactive (Line 5), while the `remaining_vacation` reactive (Line 6) tracks the number of remaining vacation days.

2.2 Interactions

Changes to the state of the system, e.g., adding appointments to a calendar, happen through explicit *interactions*. Each interaction has two sets of type parameters: the types of source reactivities that it modifies and the types of parameters that are provided when the interaction is applied. For example, the `add_appointment` interaction in Line 8 modifies a reactive of type `Calendar` and takes a parameter of type `Appointment`. The semantics of an interaction `I` are defined in four parts: (1) `requires` (Line 9) defines the preconditions that must hold for `I` to be executed, (2) `executes` (Line 11) defines the changes to source reactivities, (3) `ensures` (Line 12) defines the postconditions that must hold at the end of `I`’s execution, (4) `modifies`



■ **Figure 1** Concurrent execution of interactions may cause invariant violations. In this example, device D_1 adds a vacation of 20 days to the calendar, while D_2 concurrently adds a vacation of 12 days. Given a total amount of 30 available vacation days, this leads to a negative amount of remaining vacation once the devices synchronize.

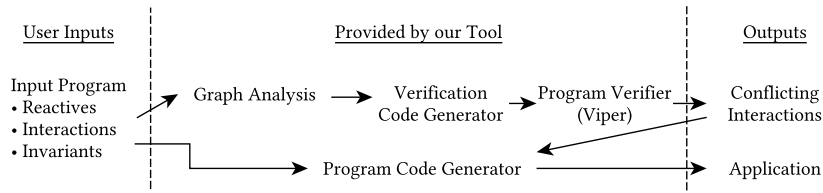
(Line 13) defines the source reactivities that I changes. The parameters of `requires`, `executes`, and `ensures` are functions that take the modified reactivities and the interaction parameters as input (`cal` is of type `Calendar` and `a` is of type `Appointment`). The splitting of the definition of interactions in four parts allows for modularization and reuse. For instance, `add_appointment` is only a partial specification of an interaction, missing the `modifies` specification. Both `add_work` (Line 15) and `add_vacation` (Line 13) specify complete interactions by adding `modifies` to `add_appointment`; they are independent interactions that differ only in their `modifies` set.

Interactions encapsulate reactions to input from the outside world (e.g., the callback in Line 18 that is triggered by the UI and applies the arguments to `add_vacation`). Applying an interaction checks the preconditions, and – if they are fulfilled – computes and applies the changes to the source reactivities, and propagates them to derived reactivities – all in a “transactional” way in the sense that all changes to affected reactivities become observable at-once (“atomically”). Only source reactivities are replicated between devices, while derived reactivities are computed by each device individually. LoRe guarantees that executing interactions does not invalidate neither postconditions nor invariants.

2.3 Invariants and Conflicts

LoRe expects the developer to use *invariants*, introduced with the keyword `invariant`, to specify application properties that should always hold. Invariants are first-order logic assertions given to a verifier based on the Viper verification infrastructure [28]. Invariants can help uncover programming bugs and reveal where the eventually-consistent replication based on CRDTs could lead to safety problems.

For illustration, consider the invariants for the calendar application in Lines 20 and 23. The invariant in Line 20 requires that all appointments must start before they end. Notice, how the invariant can be defined without knowing the amount of calendars and the actual structure of the data-flow graph by simply referring to the `all_appointments` reactive. This invariant represents a form of input validation, and is directly ensured by `add_appointment` interactions because the precondition on the arguments requires the added appointment to start before it ends (Line 9). In absence of this precondition, the LoRe compiler would reject the program and report a safety error due to a possible invariant violation. The invariant in Line 23 requires that employees do not take more vacation days than available to them. Again, this is locally enforced by the precondition of the `add_vacation` interaction, which ensures that new entries do not exceed the remaining vacation days. But there is nothing stopping two devices from concurrently adding vacation entries, which in sum violates the invariant. Figure 1 illustrates such a situation: A user plans a vacation of 20 days on the mobile phone (device D_1) and later schedules a 12-day vacation on a desktop (device D_2), at a time when D_1 was offline. Thus, both interactions happened concurrently and after merging the states the calendar contains a total of 32 days of vacation, violating the `remaining_vacation` invariant.



■ **Figure 2** Overview of LoRe’s automated compilation and verification procedure.

This example illustrates a conflict between (concurrent) execution of interactions – in this case, two executions of the `add_vacation` Interaction must be coordinated (synchronized) in order to avoid invariant violations. The LoRe compiler reports conflicting interactions to the developer and automatically synthesizes the required coordination code for the execution of such interactions (see Section 3.3). In a local-first setting, it is of paramount importance to minimize the required coordination to allow offline availability. Reporting of conflicts due to invariants helps developers to explore different situations and make informed decisions about the safety guarantees of their program. When they find that their program requires too much synchronization, they can lower the guarantees by adapting their invariants.

3 Implementation

Figure 2 depicts the architecture of LoRe’s verifying compiler. The input to the compiler is a program with its specifications expressed by the invariants, e.g., the program in Listing 1. The output consists of the conflicting interactions and a safe executable program. We use the Viper program verifier to reason about invariant violations and possible conflicts between interactions. We employ an analysis of the data-flow graph to minimize proof obligations to those invariant pairs that may actually conflict.

The rest of this section describes the pipeline from Figure 2 in detail – from left to right, top to bottom.

3.1 Graph Analysis

Checking all pairs of interactions for confluence would result in an exponential amount of proof obligations. To avoid this, we employ a graph analysis to quickly detect pairs of interactions that cannot conflict, because they change completely separate parts of the data-flow graph. For illustration, consider the `add_work` interaction. It modifies the `work` reactive, and – transitively – `all_appointments`. Hence, the reachable reactivities are $\{work, all_appointments\}$ and only the first but not the second invariant in Listing 1 overlaps. Thus, neither the `remaining_vacation` reactive, nor the invariant on this reactive will be part of the proof obligation for the `add_work` interaction.

3.2 Automated Verification

We use Viper to classify each interaction into one of the following three categories: 1) *Non-preserving* interactions can violate invariants during execution and are reported as bugs to the developer. 2) *Invariant-preserving* interactions preserve an invariant when executed on a single device but can violate an invariant in the presence of concurrent interactions by other devices. 3) *Invariant-confluent* [4] interactions can be executed concurrently without ever violating an invariant. Whenever two interactions in the second category must not

be executed concurrently to each other, they are *conflicting* and have to be coordinated to ensure invariant-safety. Using the proofs, we can precisely determine the sets of conflicting interactions and automatically synthesize coordination procedures which ensure safety at runtime while limiting the synchronization points to the necessary cases.

3.3 Synchronization at Runtime

Our compiler generates an executable application by converting the data-flow graph to a distributed REScala program [26, 27]. REScala supports all reactive features we require and integrates well with our CRDT-based replication, but has no mechanism for synchronization. LoRe’s formal synchronization semantics (see the extended version of this work²) could be implemented using any existing form of coordination, such as a central server, a distributed ledger, a consensus algorithm, or a distributed locking protocol. Which choice is suitable, depends on the target application, network size, and the reliability of the chosen transport layer. We use a simple distributed locking protocol for our implementation: Each interaction has an associated lock (represented as a simple token). Whenever a device wants to execute an interaction, it acquires the tokens of all conflicting interactions. If multiple devices request the same token concurrently, the token is given to the device with the lowest ID that requested it. This ensures deadlock freedom; fairness is left for future work. After performing the interaction, the resulting state changes are synchronized with the other devices and the tokens are made available again. Timeouts ensure that whenever a device crashes or becomes unavailable for a longer period of time, its currently owned tokens are released and any unfinished interactions by the device are aborted.

4 Evaluation

Our evaluation aims to validate two claims about LoRe’s programming model:

- C1:** It facilitates the development of safe local-first software.
- C2:** It enables an efficient and modular verification of safety properties.

We base our validation on two case studies. First, we implemented the standard TPC-C benchmark [36] as a local-first application in LoRe. This case study enables comparing LoRe’s model with traditional database-centered development of data processing software and showcasing the benefits of LoRe’s verifiable safety guarantees on standard *consistency conditions*. Second, we implemented the running calendar example (Section 2) using Yjs [31]. This case study allows comparing LoRe with an existing framework for local-first applications that we consider a representative of the state-of-the-art.

4.1 Does LoRe facilitate the development of safe local-first software?

4.1.1 Local-first TPC-C

TPC-C models an order fulfillment system with multiple warehouses in different districts, consisting of five *database transactions* alongside twelve *consistency conditions*. We implemented TPC-C in LoRe by mapping database tables to source reactivities and derived database values to derived reactivities. Each database transaction was modelled as a LoRe interaction.

While modelling the application using reactivities might require some adaption from developers not familiar with data-flow programming, we found that using derived reactivities led to a more concise and less error-prone design when compared to storing derived values in separate tables. For example, instead of storing the year to date (YTD) value of each TPC-C district in a separate table and updating it each time the payment history changes,

■ **Listing 2** Defining source and derived variables in Yjs.

```

1  const ydoc = new Y.Doc()
2  let work = ydoc.getMap('work');
3  let vacation = ydoc.getMap('
4      vacation');
5  let all_appointments;
6  let remaining_vacation = 30;
7  work.observe(ymapEvent => {
8      all_appointments = getMap(work,
9          vacation);
10 })
11 vacation.observe(ymapEvent => {
12     let days_total = getTotalVacDays
13         (vacation);
14     remaining_vacation = 30 -
15         daysTotal;
16     all_appointments = getMap(work,
17         vacation);
18 })

```

■ **Listing 3** Adding appointments in Yjs.

```

1  function addAppointment(calendar,
2      appointment) {
3      if(appointment.start <
4          appointment.end){
5          calendar.set(appointment.id,
6              appointment);
7      }
8  }
9  function addVacation(appointment)
10 {
11     let days =
12         appointment.getDays();
13     if(remainingVacation < days){
14         console.log("Sorry, no
15             vacation left!");
16     }
17     else{
18         addAppointment(vacation,
19             appointment)
20     }
21 }

```

we can model the district YTD as a derived reactive. Following this approach automatically guarantees 9 out of 12 consistency conditions of TPC-C that express consistency requirements between multiple related tables. We were able to phrase the remaining 3 conditions as invariants by directly translating the natural language formulations into logical specifications. To prove them, we additionally needed to specify pre- and postconditions of interactions corresponding to transactions. Other than that, LoRe relieves the TPC-C developer from any considerations of transaction interleavings that could potentially violate the conditions as well as from implementing the synchronization logic, both tedious and error-prone processes.

4.1.2 Yjs-based Calendar

We now compare the LoRe implementation of the distributed calendar to an implementation using the state of the art local-first framework Yjs [31]. Like other solutions for local-first software, Yjs uses a library of CRDTs (usually maps, sets, sequences / arrays, and counters) composed into nested trees – called a *documents* – used to model domain objects.

Source and Derived Variables. For illustration, consider Listing 2, showing how one could implement the domain model of the calendar application. Lines 2 and 3 initialize two CRDTs for the work and vacation calendar. Yjs has no abstraction for derived values and only provides callbacks for reacting to value changes, e.g., Lines 7-15 declare callback methods that update the derived variables in case the Yjs document changes.

Safety Guarantees. Using callbacks to model and manage complex state that changes both in time and in space has issues. It requires that developers programmatically update the derived values once the sources get updated, via local interactions or on receiving updates from other devices, with no guarantees that they do so consistently. It yields a complex control-flow and requires intricate knowledge of the execution semantics to ensure atomicity of updates, let alone to enforce application-level safety properties. Frameworks like Yjs do not offer support for application invariants and thus force developers to integrate custom safety measures at each possible source of safety violations.

■ **Table 1** Seconds to verify combinations of interactions and invariants of the two example applications. Each entry represents the mean verification time over 5 runs with the deviation shown in parentheses.

Distributed Calendar			
Interaction	Invariant		
	1	2	
Add vacation	3.32 (± 0.05)	2.97 (± 0.03)	
Remove vacation	3.28 (± 0.06)	3.00 (± 0.02)	
Change vacation	3.32 (± 0.05)	3.04 (± 0.03)	
Add work	3.31 (± 0.04)	–	
Remove work	3.30 (± 0.06)	–	
Change work	3.34 (± 0.06)	–	
TPC-C			
Interaction	Consistency Condition		
	3	5	7
New Order	45.4 (± 63.69)	7.63 (± 0.11)	14.49 (± 7.31)
Delivery	5.78 (± 0.03)	5.74 (± 0.07)	5.76 (± 0.09)

In summary, while the replication capabilities of systems like Yjs are valuable for local-first applications, these systems still require the developer to do state management manually. The prevailing use of callbacks and implicit dependencies makes reasoning about the code challenging for both developers and automatic analyses. In contrast, LoRe allows declarative definitions of derived values, with positive effects on reasoning [33, 10]. Moreover, LoRe integrates application invariants as explicit language constructs, which allows for a modular specification and verification and relieves developers from having to consider every involved interaction whenever the specification changes.

4.2 Does LoRe enable efficient and modular verification of safety properties?

To empirically evaluate the performance of LoRe’s verifier, we quantify how long it takes to verify different combinations of interactions and invariants of our two case studies. The results are shown in Table 1. The calendar example has two additional types of interactions, which we have not shown in Section 2: removing and changing calendar entries. This leads to a total of 6 interactions (3 per calendar reactive). For TPC-C we only had to verify consistency conditions 3, 5, and 7 because the others were already ensured by the respective derived reactivities. The benchmarks were performed on a desktop PC with an AMD Ryzen 7 5700G CPU and 32 GB RAM using Viper’s *silicon* verification backend (release v.23.01) [37].

Results. In summary, every interaction/invariant combination in our case studies could be verified in less than a minute. Verification times differed depending mainly on the complexity and length of the interactions and invariants under consideration. Differences become apparent especially when looking at the results for TPC-C. Proofs involving the *New Order* interaction, which is the most “write-heavy” interaction of TPC-C that changes many source reactivities at once, generally took longer to verify than others. For *New Order*, we also observe a much higher deviation of up to 64 seconds which we assume to be caused

by internal Z3 heuristics⁵. When interpreting the results, it is important to note that each interaction/invariant combination has to be verified only once and independently of other combinations. Large-scale applications can be verified step-by-step by splitting them into smaller pieces. Furthermore, we limit the need for verification to potential conflicts that we derive from the reactive data-flow graph. Programmers can add new functionality to the application (i.e., specify interactions) and only have to reason about the properties of that new functionality (i.e., specify its invariants) and the system ensures global safety – at only the cost of the amount of overlap with existing functionality. This allows for an incremental development style, where only certain parts of programs have to be (re-)verified, when they have been changed or added.

5 Related Work

Our work relates to three areas: distributed datatypes, formal reasoning, and language-based approaches. Sections below relate work from each area to respective aspects of our approach.

5.1 Consistency Through Distributed Data Types

Conflict-Free Replicated Datatypes (CRDTs) [34, 32] are a building block for constructing systems and applications that guarantee eventual consistency. CRDTs are used in distributed database systems such as *Riak* [18] and *AntidoteDB* [1]. These databases make it possible to construct applications that behave under mixed consistency, but unlike our approach, they leave reasoning about application guarantees to the programmer. Several works [12, 30] propose frameworks for formally verifying the correctness of CRDTs, while others [16, 20] focus on synthesizing correct-by construction CRDTs from specifications.

De Porre et al. [9, 8] suggest *strong eventually consistent replicated objects (SECROs)* relying on a replication protocol that tries to find a valid total order of all operations. Similar to LoRe, *SECROs* [9, 8] and *Hamsaz* [14] extend upon the eventually consistent replication of CRDTs by automatically choosing the right consistency level based on application invariants. Both approaches tie consistency and safety properties to specific datatypes/objects. This is not sufficient to guarantee end-to-end correctness of an entire local-first application - consistency bugs can still manifest in derived information (e.g., in the user interface).

5.2 Automated Reasoning about Consistency Levels

Our formalization is in part inspired by the work of Balegas et al. [6, 7] on *Indigo*. The work introduces a database middleware consisting of transactions and invariants to determine the ideal consistency level – called *explicit consistency*. They build on the notion of *invariant-confluence* for transactions that cannot harm an invariant which was first introduced by Bailis et al. [4]. While they work on a database level, we show how to integrate this reasoning approach into a programming language. An important difference between our *invariant-confluence* and the one by Balegas et al. [6] is that our approach also verifies local preservation of invariants, whereas their reasoning principle assumes invariants to always hold in a local context. In a more recent work called *IPA*, Balegas et al. [5] propose a static analysis technique that aims at automatically repairing transaction/invariant conflicts without adding synchronization between devices. We consider this latter work complementary to ours.

⁵ These could likely be improved by annotating quantifiers in invariants and pre-/postconditions with hand-crafted trigger expressions [28].

Whittaker and Hellerstein [38] also build on the idea of invariant-confluence and extend it to the concept of *segmented invariant-confluence*. Under segmented invariant-confluence, programs are separated into segments that can operate without coordination and coordination only happens in between the segments. The idea is similar to our definition of *conflicting interactions*, however, their procedure cannot suggest a suitable program segmentation, but requires developers to supply them.

The *SIEVE* framework [22] builds on the previous work on *Red/Blue-Consistency* [23] and uses invariants and program annotations to infer where a Java program can safely operate under CRDT-based replication (*blue*) and where strong consistency is necessary (*red*). They do so by relying on a combination of static and dynamic analysis techniques. Compared to *SIEVE*, our formal reasoning does not require any form of dynamic analysis. *Blazes* [2] is another analysis framework that uses programmer supplied specifications to determine where synchronization is necessary to ensure eventual consistency. Contrary to *Blazes*, LoRe ensures that programs are “by design” at least eventually consistent, while also allowing the expression and analysis of programs that need stronger consistency. *Q9* [15] is a bounded symbolic execution system, which identifies invariant violations caused by weak consistency guarantees. Similar to our work, *Q9* can determine where exactly stronger consistency guarantees are needed to maintain certain application invariants. However, its verification technique is bound by the number of possible concurrent operations. LoRe can provide guarantees for an unlimited amount of devices with an unlimited amount of concurrent operations.

5.3 Language Abstractions for Data Consistency

We categorize language-based approaches based on how they achieve consistency and on the level of programmer involvement.

Manual Choice of Consistency Levels. Li et al. [23] propose *RedBlue Consistency* where programmers manually label their operations to be either blue (eventually consistent) or red (strongly consistent). In MixT [25], programmers annotate classes with different consistency levels and the system uses an information-flow type system to ensure that the requested guarantees are maintained. However, this still requires expert knowledge about each consistency level, and wrong choices can violate the intended program semantics. Other approaches [29, 19] expect programmers to choose between *consistency* and *availability*, again leaving the reasoning duty about consistency levels to the programmer. Compared to LoRe, languages in this category place higher burden on programmers: They decide which operation needs which consistency level, a non-trivial and error-prone selection.

Automatically Deriving Consistency from Application Invariants. *CAROL* [21] uses CRDTs to replicate data and features a refinement typing discipline for expressing safety properties similar to our *invariants*. Carol makes use of pre-defined datatypes with *consistency guards* used by the type system to check for invariant violations. The compatibility of datatype operations and consistency guards is verified ahead of time using an algorithm for the Z3 SMT solver. This approach hides much of the complexity from the programmer, but the abstraction breaks once functionality that is not covered by a pre-defined datatype is needed. Unlike Carol, LoRe does not rely on predefined consistency guards, but allows the expression of safety properties as arbitrary logical formulae. Additionally, *CAROL* only checks the concurrent interactions of a program for invariant violations, whereas LoRe verifies the overall application including non-distributed parts. Sivaramakrishnan et al. [35] propose *QUELEA*, a declarative language for programming on top of eventually consistent datastores. It features a contract-language to express application-level invariants and automatically

generates coordination strategies in cases where invariants could be violated by concurrent operations. *QUELEA*'s contract-language requires programmers to express the desired properties using low-level visibility relations, which can be challenging to get right for non-experts. LoRe avoids this intermediate reasoning and automatically derives the right level of consistency for satisfying high-level safety invariants to enable end-to-end correctness.

Automating Consistency by Prescribing the Programming Model. Languages in this category seek to automate consistency decisions by prescribing a certain programming model such that certain consistency problems are impossible to occur. In *Lasp* [24], programmers model the data flow of their applications using combinator functions on CRDTs. Programs written in *Lasp* always provide eventual consistency but contrary to LoRe, *Lasp* does not allow arbitrary compositions of distributed datatypes. *Bloom* [3] provides programmers with ways to write programs that are *logically monotonic* and therefore offer automatic eventual consistency. Both *Lasp* and *Bloom*, however, are not meant to formulate programs that need stronger consistency guarantees. LoRe is similar to *Lasp* and *Bloom* in the sense that we also prescribe a specific – reactive – programming style. However, our programming model is less restrictive and allows arbitrary compositions of distributed datatypes. This is enabled by leveraging the composability properties of reactive data-flow graphs. Secondly, LoRe provides a principled way to express hybrid consistency applications with guarantees stronger than eventual consistency. Drechsler et al. [11] and Mogk et al. [26, 27] also use a reactive programming model similar to ours to automate consistency in presence of multi-threading respectively of a distributed execution setting. However, they do not support a hybrid consistency model. Drechsler et al. [11] enable strong consistency (serializability) only, while Mogk et al. [26, 27] support only eventual consistency.

6 Conclusion and Future Work

In this paper, we proposed LoRe, a language for local-first software with verified safety guarantees. *LoRe* combines the declarative data flow of reactive programming with static analysis and verification techniques to precisely determine concurrent interactions that could violate programmer-specified safety properties. We presented a formal definition of the programming model and a modular verification that detects concurrent executions that may violate application invariants. In case of invariant violation due to concurrent execution, LoRe automatically enforces the necessary amount of coordination. LoRe's verifying compiler translates LoRe programs to Viper [28] for automated verification and to Scala for the application logic including synthesized synchronization to guarantee the specified safety invariants. An evaluation of LoRe's programming model in two case studies confirms that it facilitates the development of safe local-first applications and enables efficient and modular automated reasoning about an application's safety properties. Our evaluation shows that verification times are acceptable and that the verification effort required from developers is reasonable.

In the future, it would be desirable to integrate existing libraries of verified CRDTs [12] or even solutions that allow ad-hoc verification of CRDT-like datatypes [30, 20]. This would enable us to support a wider range of data types or even allow programmers to use custom distributed datatypes, which can be verified to be eventually consistent. Furthermore, our current data-flow analysis is limited to static data-flow graphs. While static reasoning about dynamic graphs is impossible in the general case, most applications make systematic use of dynamic dependencies, and we believe it would be feasible to support common cases.

References

- 1 Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016. doi:10.1109/ICDCS.2016.98.
- 2 Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. Blazes: Coordination analysis for distributed programs. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, March 2014. doi:10.1109/ICDE.2014.6816639.
- 3 Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. Consistency Analysis in Bloom: A CALM and Collected Approach. In *CIDR*. Citeseer, 2011. URL: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf.
- 4 Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment*, 8(3):185–196, November 2014. doi:10.14778/2735508.2735509.
- 5 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases. *Proceedings of the VLDB Endowment*, 12(4):404–418, December 2018. doi:10.14778/3297753.3297760.
- 6 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, pages 1–16, 2015. doi:10.1145/2741948.2741972.
- 7 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Towards Fast Invariant Preservation in Geo-replicated Systems. *ACM SIGOPS Operating Systems Review*, 49(1):121–125, January 2015. doi:10.1145/2723872.2723889.
- 8 Kevin De Porre, Carla Ferreira, Nuno M. Preguiça, and Elisa Gonzalez Boix. Ecos: building global scale systems from sequential code. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485484.
- 9 Kevin De Porre, Florian Myter, Christophe De Troyer, Christophe Scholliers, Wolfgang De Meuter, and Elisa Gonzalez Boix. Putting order in strong eventual consistency. In *Distributed Applications and Interoperable Systems*, pages 36–56, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-22496-7_3.
- 10 Moritz Dinser. An empirical study on reactive programming. Master’s thesis, Technische Universität Darmstadt, 2021. URL: <http://tubama.ulb.tu-darmstadt.de/id/eprint/30079>.
- 11 Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-Safe Reactive Programming. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018. doi:10.1145/3276477.
- 12 Victor B F Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. Verifying Strong Eventual Consistency in Distributed Systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):109:1–109:28, October 2017. doi:10.1145/3133933.
- 13 Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. Lore: A programming model for verifiably safe local-first software, 2023. arXiv:2304.07133.
- 14 Farzin Houshmand and Mohsen Lesani. Hamsaz: Replication Coordination Analysis and Synthesis. *Proceedings of the ACM on Programming Languages*, 3(POPL):74:1–74:32, January 2019. doi:10.1145/3290387.
- 15 Gowtham Kaki, Kapil Earanky, K C Sivaramakrishnan, and Suresh Jagannathan. Safe Replication through Bounded Concurrency Verification. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), October 2018. doi:10.1145/3276534.
- 16 Gowtham Kaki, Swarn Priya, K C Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):154:1–154:29, October 2019. doi:10.1145/3360580.

- 17 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, New York, NY, USA, October 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.
- 18 Rusty Klophaus. Riak Core: Building Distributed Applications without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming*, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1900160.1900176.
- 19 Mirko Köhler, Nafise Eskandani, Pascal Weisenburger, Alessandro Margara, and Guido Salvaneschi. Rethinking Safe Consistency in Distributed Object-Oriented Programming. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):188:1–188:30, 2020. doi:10.1145/3428256.
- 20 Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: Synthesizing CRDTs with verified lifting. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):173:1349–173:1377, October 2022. doi:10.1145/3563336.
- 21 Nicholas V Lewchenko, Arjun Radhakrishna, Akash Gaonkar, and Pavol Černý. Sequential Programming for Replicated Data Stores. *Proceedings of the ACM on Programming Languages*, 3(ICFP):106:1–106:28, July 2019. doi:10.1145/3341710.
- 22 Cheng Li, João Leitão, Allen Clement, Nuno M. Pregoça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, 2014. USENIX Association. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.
- 23 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Pregoça, and Rodrigo Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 265–278, 2012. URL: <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li>.
- 24 Christopher Meiklejohn and Peter Van Roy. Lasp: A Language for Distributed, Coordination-free Programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*, pages 184–195, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2790449.2790525.
- 25 Mae Milano and Andrew C Myers. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192375.
- 26 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPICs.ECOOP.2018.1.
- 27 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A Fault-tolerant Programming Model for Distributed Interactive Applications. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):144:1–144:29, October 2019. doi:10.1145/3360570.
- 28 P Müller, M Schwerhoff, and A J Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583, Berlin, Heidelberg, 2016. Springer-Verlag. doi:10.1007/978-3-662-49122-5_2.
- 29 Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. A CAPable Distributed Programming Model. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3276954.3276957.

- 30 Sreeja S Nair, Gustavo Petri, and Marc Shapiro. Proving the Safety of Highly-Available Distributed Objects. In *Programming Languages and Systems*, pages 544–571, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-44914-8_20.
- 31 Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 19th International Conference on Supporting Group Work, Sanibel Island, FL, USA, November 13 - 16, 2016*. Association for Computing Machinery, 2016. doi:10.1145/2957276.2957310.
- 32 Nuno Preguiça. Conflict-free Replicated Data Types: An Overview. *ArXiv*, June 2018. doi:10.48550/arXiv.1806.10254.
- 33 Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering*, 43(12), 2017. doi:10.1109/TSE.2017.2655524.
- 34 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. URL: <https://hal.inria.fr/inria-00555588>.
- 35 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15*, pages 413–424, New York, NY, USA, June 2015. Association for Computing Machinery. doi:10.1145/2737924.2737981.
- 36 TPC. TPC-C Specification 5.11.0, 2021. URL: http://tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf.
- 37 Viper. Viperproject/silicon Github Repository. Viper Project, April 2021. URL: <https://github.com/viperproject/silicon>.
- 38 Michael Whittaker and Joseph M Hellerstein. Interactive Checks for Coordination Avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, September 2018. doi:10.14778/3275536.3275538.

Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises

Feiyang Jin ✉

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Lechen Yu ✉

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Tiago Cogumbreiro ✉

College of Science and Mathematics, University of Massachusetts Boston, MA, USA

Jun Shirako ✉

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Vivek Sarkar ✉

College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Abstract

Much of the past work on dynamic data-race and determinacy-race detection algorithms for task parallelism has focused on structured parallelism with fork-join constructs and, more recently, with future constructs. This paper addresses the problem of dynamic detection of data-races and determinacy-races in task-parallel programs with *promises*, which are more general than fork-join constructs and futures. The motivation for our work is twofold. First, promises have now become a mainstream synchronization construct, with their inclusion in multiple languages, including C++, JavaScript, and Java. Second, past work on dynamic data-race and determinacy-race detection for task-parallel programs does not apply to programs with promises, thereby identifying a vital need for this work.

This paper makes multiple contributions. First, we introduce a featherweight programming language that captures the semantics of task-parallel programs with promises and provides a basis for formally defining *determinacy* using our semantics. This definition subsumes functional determinacy (same output for same input) and structural determinacy (same computation graph for same input). The main theoretical result shows that the absence of data races is sufficient to guarantee determinacy with both properties. We are unaware of any prior work that established this result for task-parallel programs with promises. Next, we introduce a new Dynamic Race Detector for Promises that we call DRDP. DRDP is the first known race detection algorithm that executes a task-parallel program sequentially without requiring the serial-projection property; this is a critical requirement since programs with promises do not satisfy the serial-projection property in general. Finally, the paper includes experimental results obtained from an implementation of DRDP. The results show that, with some important optimizations introduced in our work, the space and time overheads of DRDP are comparable to those of more restrictive race detection algorithms from past work. To the best of our knowledge, DRDP is the first determinacy race detector for task-parallel programs with promises.

2012 ACM Subject Classification Software and its engineering → Software creation and management; Software and its engineering → Software verification and validation; Software and its engineering → Software defect analysis; Software and its engineering → Software testing and debugging; Software and its engineering → Software notations and tools; Software and its engineering → General programming languages; Software and its engineering → Concurrent programming languages

Keywords and phrases Race detection, Promise, Determinism, Determinacy-race, Serial projection

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.13

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.24>

Funding This material is based upon work supported by the National Science Foundation under Grant No. 2204986.



© Feiyang Jin, Lechen Yu, Tiago Cogumbreiro, Jun Shirako, and Vivek Sarkar; licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 13; pp. 13:1–13:30



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



```

1  x ← alloc
2  y ← new_promise
3  async {
4    store 5 to x
5    y.set
6    return
7  }
8  a ← load x
9  y.get
10 b ← load x
11 return

```

■ **Figure 1** Example program.

1 Introduction

In recent years, promises have been incorporated as a general synchronization construct into multiple mainstream languages, including C++ [17], Java [28], and JavaScript [27]. A promise is a wrapper for a data payload that is initially empty. It typically has two operations which we refer to as `set` and `get`. Each `get` operation blocks until the promise receives a value for its payload; multiple `get` operations may be performed on the same promise from multiple tasks, and they all return the same value. Figure 1 shows the usage: an `async` task sets the promise at line 5, and the main task gets the promise at line 9.

A promise with a set payload is referred to as a *fulfilled promise*. Following standard conventions, we assume that promises obey the single-assignment policy, where an invocation of `set` on a fulfilled promise (i.e., a second assignment) will induce a runtime error. Compared to futures [20], promises generalize the semantics for synchronization in that a promise need not be bound to the return value of a specific task; instead, any task can choose to perform a `set` operation on a given promise. Promises support arbitrary point-to-point synchronization wherein one or more tasks can await the arrival of a payload for which the producer task is not known in advance. However, it has been observed that the convenience of this generality may also be accompanied by the increasing complexity of dynamic analysis for bug detection in parallel programs [42].

As with any source of parallelism, accesses to shared memory locations must be correctly ordered to avoid *determinacy races* [15], defined as race conditions causing non-determinism. A determinacy race often results from a *data race* [26,31], which occurs when two concurrent memory accesses operate on the same memory location and at least one of them is a write. A key result in our paper is that the absence of data races is sufficient to guarantee determinacy for task-parallel programs with promises; in contrast, this property does not hold for programs that use mutual exclusion constructs such as locks or transactions.

For dynamic race detectors, enumerating all possible inputs is usually intractable; therefore, they are typically *per-input* or *per-schedule* race detectors. Per-input race detectors report all potential races for a given input by covering all possible thread schedules [15,30,31,35,43–45], whereas per-schedule race detectors only cover the observed schedule when analyzing a program’s execution [1,16,33]. However, prior work related to dynamic determinacy race detection has some major limitations:

1. None of these race detectors support promises.
2. No formal definition of “determinacy” is provided. For example, in the SP-Bags paper [15], the authors state “[determinacy race]... may cause the program to behave nondeterministically. That is, different runs of the same program may produce different behaviors”. However, no formal definition was provided for what is meant by “different behaviors.”
3. It has been observed in some past work (e.g., [7, 31, 35]) that for certain classes of task-parallel programs, data-race freedom leads to determinacy but no formal proof was given for this observation.

In this paper, we introduce a featherweight programming language that captures the semantics of Task-Parallel Programs with Promises (TP3) and use it as a basis for formally defining determinacy, along with a proof that data-race freedom implies both structural and functional determinacy for TP3. We also designed and implemented a race detector for TP3. A major obstacle is that the tasks issuing `get/set` operations on a given promise cannot be identified in advance. As a result, TP3 do not satisfy the *serial-projection* property [32], i.e., the property that a sequential execution of the program with all parallel constructs removed is guaranteed to be a legal execution of the original parallel program. This feature is utilized by a number of dynamic race detectors [4, 15, 30, 35, 37]. Without this property, the sequential execution of a parallel program may be blocked by a `get` operation. To address this and other challenges, we extended the Habanero-C/C++ library [8] to enable correct single-worker execution of such programs via cooperative task switching. However, keeping track of happens-before relationships also becomes more challenging in the presence of task switching. Efficient data structures used previously [4, 15, 30, 35, 37] rely on the serial-projection property to maintain happens-before information correctly and cannot directly be used for programs with promises.

In summary, the key contributions of this paper are as follows:

1. A formalization of determinacy for TP3 using a featherweight programming language (Section 2).
2. A proof that TP3 are guaranteed to be determinate in the absence of data races (Section 3).
3. A new dynamic race detection algorithm for TP3 called Determinacy Race Detector for Promises (DRDP, Section 4). To the best of our knowledge, this is the first precise and efficient per-input race detector for TP3.
4. An implementation of DRDP on top of the Habanero-C/C++ library and its evaluation on a set of benchmarks using promises (Section 5). The results show that, with some essential optimizations introduced in our work, the space and time overheads of DRDP are comparable to those of more restrictive race detection algorithms from past work.

2 A Featherweight Language for Task-Parallel Programs with Promises

In this section, we introduce a featherweight language that features task parallelism and promises to establish determinacy. Our language is Turing-complete. We do not include functions and types because we aim to use this core language to prove properties of dynamic program executions that are possible from a given static program, rather than using this language for static program analysis.

Program	P	::=	$s ; P \mid \text{return}$	
Statement	s	::=	m $\mid \text{async}\{P\}$ (create asynchronous task) $\mid x \leftarrow \text{new_promise}$ $\mid x.\text{set}$ $\mid x.\text{get}$ $\mid x \leftarrow e$ $\mid \text{while } (0 \neq \text{load } y) \{P\}$	
Memory Operation	m	::=	$y \leftarrow \text{alloc}$ $\mid x \leftarrow \text{load } y$ $\mid \text{store } e \text{ to } y$	
Expression	e	::=	x (local variable) $\mid y$ (local variable that saves shared variable name) $\mid c$ (integer constant) $\mid e_1 + e_2$	
Runtime value	k	::=	$c \mid r \mid p$	

■ **Figure 2** Syntax of the core language.

2.1 Language Syntax

Figure 2 lists the syntax of our language. A program consists of a sequence of statements terminated by a `return`. A single statement can be task creation (`async`)¹, a memory operation (`alloc`, `load`, and `store`), a promise operation (`new_promise`, `set`, `get`), a local variable assignment (`x ← e`), or a conditional loop (`while`). We use symbols x and y to denote local variables within a specific task, along with an LLVM-style syntax and convention.

The language syntax introduces two types of variables: local variables and shared variables. Each task has a set of scoped local variables which must satisfy the single-assignment rule; apart from local variables, tasks may also access shared variables using memory operations. In fact, the only way for two tasks² to share data is via memory operations.

Shared variables are modeled as a global map (i.e., *memory*), in which each instance has a unique name assigned during its allocation; this name serves as a *reference* for operating on the corresponding shared variable. Statement `y ← alloc` allocates a new shared variable, with its reference saved into y . Statement `x ← load y` retrieves a shared variable’s content using the reference in local variable y and saves the retrieved content into another local variable x . Likewise, statement `store e to y` locates a shared variable using the reference in y and updates the shared variable’s content with value e .

Statement `x ← e` initializes single-assignment local variable x with the value e . According to the syntax, e could be an integer, a local variable, or a sum of two expressions. Statement `async{P}` spawns a concurrent task to execute the body P . Statement `while (0 ≠ load y) {P}` will continuously execute the loop body P until the condition no longer holds. Statement `x ← new_promise` creates a new promise, and saves a reference to the promise in local variable x . Statement `x.set` signals promise x , and statement `x.get` blocks until a task issues a `set` operation on promise x . To simplify the presentation, we eschew the data payload of promises, thus only offering synchronization functionality. Communicating data through a promise is still possible, but must be encoded using additional shared variables.

¹ We do not include join operations like `finish` and `sync` in our language, since they can be modeled using sets of promises.

² For convenience, the main program is considered to be a root task.

2.2 Runtime State

The runtime state σ of our language is a pair $\sigma = (M, G)$, where M maps shared variables into runtime values, and a *computation graph* G . We denote a map M as $\{r_1 : k_1, r_2 : k_2, \dots, r_n : k_n\}$. We use the notation $M[r := k]$ to extend M . We define computation graphs G inductively using the rules in Figure 3.

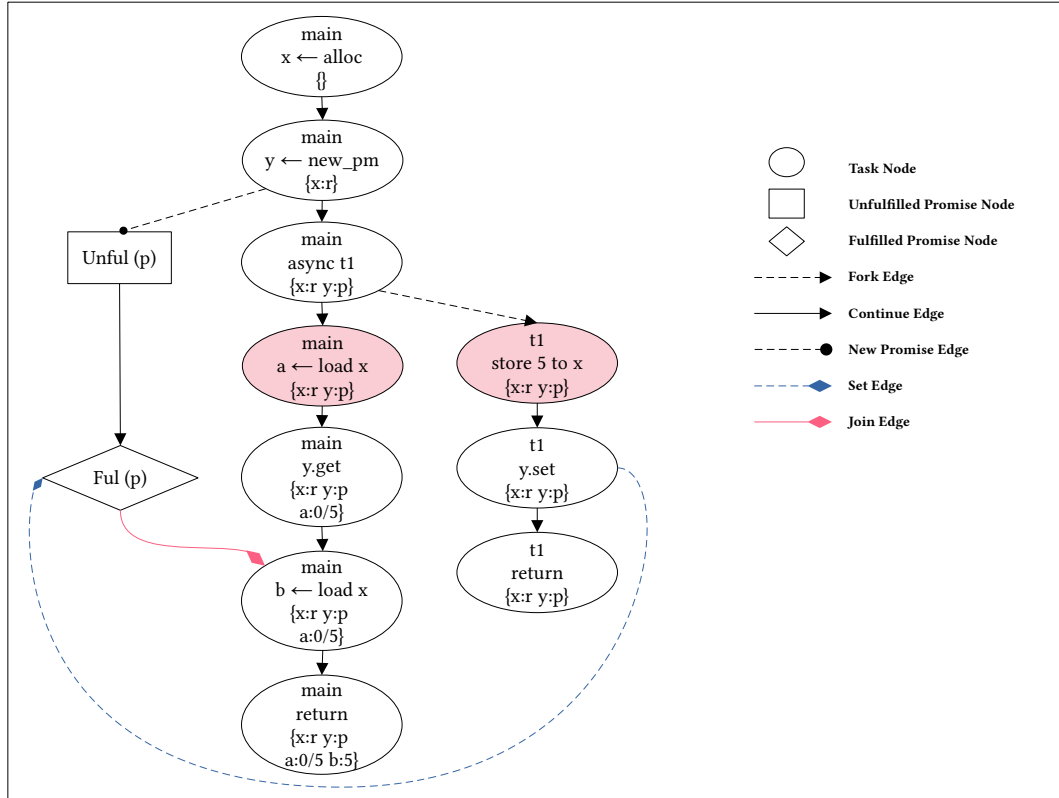
$n ::= t \mid \text{Unful}(p) \mid \text{Ful}(p)$	Runtime nodes
$t ::= [f, P, L]$	Task nodes
$G ::= t$	
$\mid G + t_1 \xrightarrow{\text{cont}} t_2$	Continue edges
$\mid G + t_2 \xleftarrow{\text{fork}} t_1 \xrightarrow{\text{cont}} t_3$	Fork edges
$\mid G + \text{Unful}(p) \xleftarrow{\text{np}} t_1 \xrightarrow{\text{cont}} t_2$	New promise edges
$\mid G + \text{Unful}(p) \xrightarrow{\text{cont}} \text{Ful}(p) \xleftarrow{\text{set}} t_1 \xrightarrow{\text{cont}} t_2$	Set edges
$\mid G + t_1 \xrightarrow{\text{cont}} t_2 \xleftarrow{\text{join}} \text{Ful}(p)$	Join edges
$\text{R-NAT} \frac{}{c \Downarrow_{t_{mem}} c}$	
$\text{R-ADD} \frac{e_1 \Downarrow_{t_{mem}} c_1 \quad e_2 \Downarrow_{t_{mem}} c_2}{e_1 + e_2 \Downarrow_{t_{mem}} c_1 + c_2}$	
$\text{R-VAR} \frac{}{x \Downarrow_{t_{mem}} t_{mem}[x]}$	

■ **Figure 3** Computation graph and reduction rule.

In a computation graph G , nodes represent the states of tasks and promises across time, and edges represent happens-before relations. There are three types of nodes in a computation graph: a *task node* t , an unfulfilled promise $\text{Unful}(p)$, and fulfilled promise $\text{Ful}(p)$. A task node t is a triple that includes a task name f , a program P to execute, and a map L representing the task's local variables. We define three helper functions to obtain the content of a task node t : t_{tid} , t_{code} , and t_{mem} , which return the task name ($t.f$), the program ($t.P$), and the local variable map ($t.L$), respectively.

The base case of a computation graph is a task node t . Notation $G + t_1 \xrightarrow{\text{cont}} t_2$ depicts a computation graph acquired by adding a new continue edge from t_1 to t_2 on top of the original graph G . Notation $G + t_2 \xleftarrow{\text{fork}} t_1 \xrightarrow{\text{cont}} t_3$ captures the semantics of task creation: node t_1 issues the task creation, node t_2 represents the child task, and node t_3 represents the continuation after spawning t_2 . Since t_2 and t_3 may happen in parallel, there exists no path between the two nodes. Notation $G + \text{Unful}(p) \xleftarrow{\text{np}} t_1 \xrightarrow{\text{cont}} t_2$ represents the semantics of promise creation. Node $\text{Unful}(p)$ represents the spawned instance of promise and t_2 represents the continuation after promise creation. Notation $G + \text{Unful}(p) \xrightarrow{\text{cont}} \text{Ful}(p) \xleftarrow{\text{set}} t_1 \xrightarrow{\text{cont}} t_2$ depicts an invocation of **set** issued by node t_1 . Node $\text{Ful}(p)$ represents the fulfilled promise after the **set**, and t_2 represents the task state upon the **set** operation. Finally, Notation $G + t_1 \xrightarrow{\text{cont}} t_2 \xleftarrow{\text{join}} \text{Ful}(p)$ signifies a **get** operation. Node t_1 issues the synchronization, while node t_2 observes the synchronization.

Example. The corresponding computation graph of the example program in Figure 1 is shown in Figure 4. Note that there is a determinacy race between line 4 and line 8 in the program. In different executions, variable a can be either 0 or 5. The race is also reflected in the computation graph, as no path connects the two nodes in red. On the other hand, the store in line 4 happens before the load on line 10 because of the `get` operation in line 9; the final value of variable b will always be five, regardless of the actual task schedule.



■ **Figure 4** Associated computation graph of Figure 1.

Leveraging the formal definition in Figure 3, we formalize the happens-before relation between two task nodes and memory accesses performed by task nodes.

► **Definition 1** (Happens-before). We say node v precedes or happens before node u if and only if one directed path from v to u exists in the computation graph. We denote the happens-before relation as $v \rightsquigarrow u$. We use $v \not\rightsquigarrow u$ to indicate that there exists no path from u to v .

► **Definition 2** (May-happen-in-parallel). Node v may happen in parallel with node u , denoted by $v \parallel u$, iff $u \not\rightsquigarrow v$ and $v \not\rightsquigarrow u$.

► **Definition 3** (Read and Write). A task node t reads from a shared variable r if 1). $t_{code} = \{x \leftarrow \text{load } y ; P\}$, $t_{mem}[y] = r$, or 2). $t_{code} = \{\text{while } (0 \neq \text{load } y) \{P'\} ; P\}$, $t_{mem}[y] = r$. A task node t writes to a shared variable r if $t_{code} = \{\text{store } x \text{ to } y ; P\}$ and $t_{mem}[y] = r$. Node t accesses a shared variable r if node t reads from or writes to r .

- (1) G-ASYNC $\frac{t_{code} = \text{async}\{P'\}; P \quad g \text{ does not occur in } G \quad t_{tid} = f}{(M, G) \rightarrow (M, G + [g, P', t_{mem}] \xleftarrow{\text{fork}} t \xrightarrow{\text{cont}} [f, P, t_{mem}])}$
- (2) G-ALLOC $\frac{t_{code} = y \leftarrow \text{alloc}; P \quad r \notin M \quad t_{tid} = f}{(M, G) \rightarrow (M[r := 0], G + t \xrightarrow{\text{cont}} [f, P, t_{mem}[y := r]])}$
- (3) G-LOAD $\frac{t_{code} = x \leftarrow \text{load } y; P \quad t_{mem}[y] = r \quad M[r] = k \quad t_{tid} = f}{(M, G) \rightarrow (M, G + t \xrightarrow{\text{cont}} [f, P, t_{mem}[x := k]])}$
- (4) G-STORE $\frac{t_{code} = \text{store } e \text{ to } y; P \quad e \Downarrow_{t_{mem}} k \quad t_{mem}[y] = r \quad t_{tid} = f}{(M, G) \rightarrow (M[r := k], G + t \xrightarrow{\text{cont}} [f, P, t_{mem}])}$
- (5) G-PROMISE $\frac{t_{code} = x \leftarrow \text{new_promise}; P \quad \text{Unful}(p) \notin G \quad t_{tid} = f}{(M, G) \rightarrow (M, G + \text{Unful}(p) \xleftarrow{\text{np}} t \xrightarrow{\text{cont}} [f, P, t_{mem}[x := p]])}$
- (6) G-SET $\frac{t_{code} = x.\text{set}; P \quad t_{mem}[x] = p \quad \text{Unful}(p) \text{ no outgoing edges in } G \quad t_{tid} = f}{(M, G) \rightarrow (M, G + \text{Unful}(p) \xrightarrow{\text{cont}} \text{Ful}(p) \xleftarrow{\text{set}} t \xrightarrow{\text{cont}} [f, P, t_{mem}])}$
- (7) G-GET $\frac{t_{code} = x.\text{get}; P \quad t_{mem}[x] = p \quad \text{Ful}(p) \in G \quad t_{tid} = f}{(M, G) \rightarrow (M, G + t \xrightarrow{\text{cont}} [f, P, t_{mem}] \xleftarrow{\text{join}} \text{Ful}(p))}$
- (8) G-ASSIGN $\frac{t_{code} = x \leftarrow e; P \quad e \Downarrow_{t_{mem}} k \quad t_{tid} = f}{(M, G) \rightarrow (M, G + t \xrightarrow{\text{cont}} [f, P, t_{mem}[x := k]])}$
- (9) G-WHILE-1 $\frac{t_{code} = \text{while } (0 \neq \text{load } y) \{P'\}; P \quad M[t_{mem}[y]] = 0 \quad t_{tid} = f}{(M, G) \rightarrow (M, G + t \xrightarrow{\text{cont}} [f, P, t_{mem}])}$
- (10) G-WHILE-2 $\frac{t_{code} = \text{while } (0 \neq \text{load } y) \{P'\}; P \quad M[t_{mem}[y]] \neq 0 \quad t_{tid} = f}{(M, G) \rightarrow (M, G + t \xrightarrow{\text{cont}} [f, P'; \text{while } (0 \neq \text{load } y) \{P'\}; P, t_{mem}])}$

■ **Figure 5** Small-step semantics.

2.3 Small-step Operational Semantics

We introduce the small-step operational semantics, denoted by $\sigma \rightarrow \sigma'$, in Figure 5. Spawning a task g with `async` creates a new node for the child task, which inherits the local memory of the parent task f (Rule 1). Memory allocation creates a new shared variable r , initializes it with 0, and assigns its name r to y in the local memory (Rule 2). A load retrieves the content of shared variable r from the shared memory M . A store writes the value e to shared variable r (variable name r is stored in local variable y). Creating a promise adds a new node that marks promise p as unfulfilled (Rule 5). Our semantics only allows a single `set` per promise; thus, a pre-condition of Rule 6 is to ensure that promise p is unfulfilled. Getting a promise links the fulfilled promise $\text{Ful}(p)$ to the continuation node $[f, P, t_{mem}]$, thus adding a

happens-before relation from the `set` to the `get` (Rule 7). Assignment extends local memory with a new variable x , which has the value of evaluating expression e (Rule 8). Rules 9 and 10 handle a while loop in a standard way.

Let $\text{root}(P)$ denotes runtime state $(m, [main, P, l])$: a runtime state that holds the initial memory m and the initial computation graph with a single vertex $[main, P, l]$, where l is the local memory for the single vertex. When it is clear from the context we may say P to signify the runtime state $\text{root}(P)$. Let notation $P \Downarrow \sigma$ be defined as $\text{root}(P) \rightarrow^* \sigma$ and $\sigma \not\Downarrow \sigma'$ for any σ' .

We use a naming convention that gives *unique* and *consistent* name to each variable in the operational semantics. “unique” means whenever a new task, variable, or promise is being created, it will be given a unique name that does not yet exist in σ ; “consistent” means the name is the same name that other program runs will have when creating this new task, variable or promise.

3 Proof of Determinism

In this section, we show that determinacy-race freedom implies determinism for programs in our featherweight language. Our proof structure is adapted from prior work on Concurrent Collection (CnC) [6, Theorem 1]. CnC applies a single-assignment policy on all shared variables (called *data collections* in CnC) to assure determinism. An important distinction between this work and [6] is that our formalism expresses both promises and shared memory, whereas [6] only expresses a construct akin to promises. Our proof utilizes the property of determinacy-race freedom to show that the shared memory and the computation graph will be determinate for a given input (the initial program state $\text{root}(P)$).

► **Definition 4** (Determinacy Race and Determinacy-Race Freedom). A *determinacy race* is a triple (r, t_1, t_2) ; it happens on a shared variable r if and only if two task nodes t_1 and t_2 access r , at least one of them conducts a write, and $t_1 \parallel t_2$. A computation graph G is *determinacy-race-free* if and only if for any task nodes t_1, t_2 in G , there is no determinacy race (r, t_1, t_2) for any shared variable r . A program P is *determinacy-race-free* if for any G and M such that $P \rightarrow^* (M, G)$, G is determinacy-race-free.

We define an ordering \leq on runtime states such that $\sigma \leq \sigma'$ if and only if $\text{dom}(\sigma.M) \subseteq \text{dom}(\sigma'.M)$ and $\sigma.G$ is a subgraph of $\sigma'.G$. Next, we establish the necessary lemmas to prove our main result: if $P \Downarrow \sigma$ and $P \Downarrow \sigma'$, then $\sigma = \sigma'$.

► **Lemma 5** (Monotonicity). *If $\sigma \rightarrow \sigma'$, then $\sigma \leq \sigma'$*

Proof. The proof follows by case analysis on the derivation of $\sigma \rightarrow \sigma'$. Let $\sigma = (M, G)$. The key insight is that nodes and edges are only added to G ; nodes and edges are never removed. Similarly, the domain of M either grows or remains the same. We omit the proof details. ◀

We use $\sigma \rightarrow_v \sigma'$ to denote that executing node v triggers the state transition.

► **Lemma 6** (Independence). *Let $\sigma \rightarrow_v \sigma'$, $\sigma \rightarrow_u \sigma''$ and $\sigma' \neq \sigma''$. We have $v \parallel u$.*

Proof. If $v \rightsquigarrow u$ or $u \rightsquigarrow v$, we can only derive either σ' or σ'' from σ , but not both. ◀

The next Lemma proves what we call strong local confluence. This property essentially implies that from the same program state σ , if there exists more than one choice to proceed, those different choices will eventually proceed to the same state σ_c . The proof also reveals why determinacy-race freedom is necessary for this property.

► **Lemma 7** (Strong Local Confluence). *Let P be determinacy-race-free and $P \rightarrow^* \sigma$. If $\sigma \rightarrow_v \sigma'$ and $\sigma \rightarrow_u \sigma''$, then there exists σ_c, i, j such that $\sigma' \rightarrow^i \sigma_c$, $\sigma'' \rightarrow^j \sigma_c$, $i \leq 1$ and $j \leq 1$.*

Proof. If $v = u$, we have $\sigma' = \sigma''$; in this case $\sigma_c = \sigma'$, $i = 0, j = 0$. If $v \neq u$, we claim $\sigma' \rightarrow_u \sigma_c, \sigma'' \rightarrow_v \sigma_c, i = 1, j = 1$. To prove the claim, we do a case analysis on the rule used to derive $\sigma \rightarrow_v \sigma'$.

- (3) G-LOAD: We know $v_{code} = x \leftarrow \text{load } y; P'$ and $v_{mem}[y] = r$ and $M[r] = k$. We have $\sigma'.G = \sigma.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$, $\sigma'.M = \sigma.M$. Let us do a case analysis of the rule used to derive $\sigma \rightarrow_u \sigma''$.
 1. G-ALLOC: We know $u_{code} = y' \leftarrow \text{alloc}; P''$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}[y' := r'], P'']$, $\sigma''.M = \sigma.M[r' := 0]$. Because our naming system is unique, we have $r' \neq r$. We can pick σ_c such that $\sigma_c.M = \sigma''.M$, $\sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$. It is clear that $\sigma'' \rightarrow_v \sigma_c$. It is also true that $\sigma' \rightarrow_u \sigma_c$ because our naming system is consistent among different execution. In this case, $i = 1, j = 1$.
 2. G-LOAD: $u_{code} = x' \leftarrow \text{load } y'; P''$ and $u_{mem}[y'] = r'$ and $M[r'] = k'$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}[x' := k'], P'']$, $\sigma''.M = \sigma.M$. In this case, it is fine that $r' = r$ because concurrent reads on the same memory location are allowed. We can pick σ_c such that $\sigma_c.M = \sigma''.M$, $\sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$. It is clear that $\sigma' \rightarrow_u \sigma_c$ and $\sigma'' \rightarrow_v \sigma_c$. In this case, $i = 1, j = 1$.
 3. G-STORE: We know $u_{code} = \text{store } e' \text{ to } y'; P''$ and $e' \Downarrow k'$ and $u_{mem}[y'] = r'$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P'']$, $\sigma''.M = \sigma.M[r' := k']$. If $r' = r$, this is a determinacy race by Definition 4 and Lemma 6. Because $r' \neq r$, we can pick σ_c such that $\sigma_c.M = \sigma''.M$, $\sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}[x := k], P']$. It is clear that $\sigma' \rightarrow_u \sigma_c$ and $\sigma'' \rightarrow_v \sigma_c$. In this case, $i = 1, j = 1$.
 4. Any other rules: we omit the details for other rules because the proof is similar to the above cases.
- (4) G-STORE: We know $v_{code} = \text{store } e \text{ to } y; P'$ and $e \Downarrow k$ and $v_{mem}[y] = r$. We have $\sigma'.G = \sigma.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$, $\sigma'.M = \sigma.M[r := k]$. Let us do a case analysis of the rule used to derive $\sigma \rightarrow_u \sigma''$.
 1. G-STORE: We know $u_{code} = \text{store } e' \text{ to } y'; P''$ and $e' \Downarrow k'$ and $u_{mem}[y'] = r'$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P'']$, $\sigma''.M = \sigma.M[r' := k']$. If $r' = r$, this is a determinacy race by Definition 4 and Lemma 6. Because $r' \neq r$, we can pick σ_c such that $\sigma_c.M = \sigma''.M[r := k]$, $\sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \rightarrow_u \sigma_c$ and $\sigma'' \rightarrow_v \sigma_c$. In this case, $i = 1, j = 1$.
 2. G-WHILE-*: We know $u_{code} = \text{while } (0 \neq \text{load } y') \{P'''\}; P''$ and $u_{mem}[y'] = r'$. If $r' = r$, this is a determinacy race by Definition 4 and Lemma 6. Because $r' \neq r$, we can pick σ_c such that $\sigma_c.M = \sigma''.M[r := k]$, $\sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \rightarrow_u \sigma_c$ and $\sigma'' \rightarrow_v \sigma_c$. In this case, $i = 1, j = 1$.
 3. Any other rules: we omit the details for other rules because the proof is similar to the above cases.
- (6) G-SET: we know $v_{code} = x.\text{set}; P'$ and $v_{mem}[x] = p$ and $\text{Unful}(p)$ has no outgoing edge. We have $\sigma'.G = \sigma.G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$, $\sigma'.M = \sigma.M$. Let us do a case analysis of the rule used to derive $\sigma \rightarrow_u \sigma''$.
 1. G-Set: we know $u_{code} = x'.\text{set}; P''$ and $u_{mem}[x'] = p'$ and $\text{Unful}(p')$ has no outgoing edge. We have $\sigma''.G = \sigma.G + \text{Unful}(p') \xrightarrow{cont} \text{Ful}(p') \xleftarrow{set} u \xrightarrow{cont} [u_{tid}, u_{mem}, P'']$, $\sigma''.M = \sigma.M$. If $p = p'$, this violates the single set policy for promises. Because $p \neq p'$, we can pick σ_c such that $\sigma_c.M = \sigma''.M$, $\sigma_c.G = \sigma''.G + \text{Unful}(p) \xrightarrow{cont} \text{Ful}(p) \xleftarrow{set} v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \rightarrow_u \sigma_c$ and $\sigma'' \rightarrow_v \sigma_c$. In this case, $i = 1, j = 1$.

13:10 Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises

2. G-GET: we know $u_{code} = x'.get ; P''$ and $u_{mem}[x'] = p'$ and $Ful(p') \in G$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P''] \xleftarrow{join} Ful(p'), \sigma''.M = \sigma.M$. In this case, we must have $p \neq p'$ because otherwise, we cannot make the step $\sigma \rightarrow_u \sigma'$ until v is executed. We can pick σ_c such that $\sigma_c.M = \sigma''.M, \sigma_c.G = \sigma''.G + Unful(p) \xrightarrow{cont} Ful(p) \xleftarrow{set} v \xrightarrow{cont} [v_{tid}, v_{mem}, P']$. It is clear that $\sigma' \rightarrow_u \sigma_c$ and $\sigma'' \rightarrow_v \sigma_c$. In this case, $i = 1, j = 1$.
 3. Any other rules: we omit the details for other rules because the proof is similar to the above cases.
- (7) G-GET: we know $v_{code} = x.get ; P'$ and $v_{mem}[x] = p$ and $Ful(p) \in G$. We have $\sigma'.G = \sigma.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P'] \xleftarrow{join} Ful(p), \sigma'.M = \sigma.M$. Let us do a case analysis of the rule used to derive $\sigma \rightarrow_u \sigma''$.
1. G-GET: we know $u_{code} = x'.get ; P''$ and $u_{mem}[x'] = p'$ and $Ful(p') \in G$. We have $\sigma''.G = \sigma.G + u \xrightarrow{cont} [u_{tid}, u_{mem}, P''] \xleftarrow{join} Ful(p'), \sigma''.M = \sigma.M$. In this case, it is fine if $p = p'$ because concurrent `get` operations performed on the same promise is allowed. We can pick σ_c such that $\sigma_c.M = \sigma''.M, \sigma_c.G = \sigma''.G + v \xrightarrow{cont} [v_{tid}, v_{mem}, P'] \xleftarrow{join} Ful(p)$. It is clear that $\sigma' \rightarrow_u \sigma_c$ and $\sigma'' \rightarrow_v \sigma_c$. In this case, $i = 1, j = 1$.
 2. Any other rules: we omit the details for other rules because the proof is similar to the above cases.
- Any other rules v and u could execute: we omit the details for other rules because the proof is similar to the above cases. ◀

► **Lemma 8** (Strong One-Sided Confluence). *Let P be determinacy-race-free and $P \rightarrow^* \sigma$. If $\sigma \rightarrow \sigma', \sigma \rightarrow^m \sigma''$, where $1 \leq m$, then there exist σ_c, i, j such that $\sigma' \rightarrow^i \sigma_c, \sigma'' \rightarrow^j \sigma_c, i \leq m$ and $j \leq 1$.*

Proof. We prove it by inducting on m .

Base case: $m = 1$. Proved by Lemma 7.

Induction step: suppose $\sigma \rightarrow^m \sigma'' \rightarrow \sigma'''$. By our induction hypothesis, the lemma holds for m . We have σ'_c, i', j' such that $\sigma' \rightarrow^{i'} \sigma'_c$ and $\sigma'' \rightarrow^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq 1$. We want to prove the lemma holds for $m + 1$. There are two cases based on the value of j' :

1. $j' = 0$. In this case, $\sigma'_c = \sigma''$. We pick $\sigma_c = \sigma''', i = i' + 1, j = 0$. Because $i' \leq m$, we have $i \leq m + 1$, which obeys the lemma.
2. $j' = 1$. In this case, we have $\sigma'' \rightarrow \sigma'''$ and $\sigma'' \rightarrow \sigma'_c$. By Lemma 7, there exist σ_d, a, b such that $\sigma''' \rightarrow^a \sigma_d$ and $\sigma'_c \rightarrow^b \sigma_d$ and $a \leq 1$ and $b \leq 1$. So we also have $\sigma' \rightarrow^{i'} \sigma'_c \rightarrow^b \sigma_d$. As a result, we pick $\sigma_c = \sigma_d, i = i' + b, j = a$. This is fine because $i = i' + b \leq m + 1$ and $j = a \leq 1$. ◀

► **Lemma 9** (Strong Confluence). *Let P be determinacy-race-free and $P \rightarrow^* \sigma$. If $\sigma \rightarrow^n \sigma', \sigma \rightarrow^m \sigma''$, where $1 \leq m, 1 \leq n$, then there exist σ_c, i, j such that $\sigma' \rightarrow^i \sigma_c, \sigma'' \rightarrow^j \sigma_c, i \leq m$ and $j \leq n$.*

Proof. We prove it by inducting on n .

Base case: $n = 1$. Proved by Lemma 8.

Induction step: suppose $\sigma \rightarrow^n \sigma' \rightarrow \sigma'''$. By our induction hypothesis, the lemma holds for n . We have σ'_c, i', j' such that $\sigma' \rightarrow^{i'} \sigma'_c$ and $\sigma'' \rightarrow^{j'} \sigma'_c$ and $i' \leq m$ and $j' \leq n$. We want to prove the lemma holds for $n + 1$. There are two cases based on the value of i' :

1. $i' = 0$. In this case, $\sigma' = \sigma'_c$. We pick $\sigma_c = \sigma''', i = 0, j = j' + 1$. Because $j' \leq n$, we have $j \leq n + 1$, which obeys the lemma.

2. $i' \geq 1$. In this case, we have $\sigma' \rightarrow \sigma'''$ and $\sigma' \rightarrow^{i'} \sigma'_c$. By Lemma 8, there exist σ_d, a, b such that $\sigma''' \rightarrow^a \sigma_d$ and $\sigma'_c \rightarrow^b \sigma_d$ and $a \leq i'$ and $b \leq 1$. So we also have $\sigma'' \rightarrow^{j'} \sigma'_c \rightarrow^b \sigma_d$. As a result, we pick $\sigma_c = \sigma_d, i = a, j = j' + b$. This is fine because $i = a \leq i' \leq m$ and $j = j' + b \leq n + 1$. ◀

► **Lemma 10** (Confluence). *Let P be determinacy-race-free and $P \rightarrow^* \sigma$. If $\sigma \rightarrow^* \sigma'$ and $\sigma \rightarrow^* \sigma''$, then there exists σ_c such that $\sigma' \rightarrow^* \sigma_c$ and $\sigma'' \rightarrow^* \sigma_c$.*

Proof. Implied by Lemma 9 ◀

With Lemma 10, we are ready to present our main theorem. Notice that we do not assume deadlock freedom for the program P . The notation $P \Downarrow \sigma$ is defined as P cannot make further progress; some `get` operations never resume after blocking because no task elects to set the required promises. However, our main theorem reveals that even if deadlock(s) exists in a determinacy-race-free program P , any execution of P will still reach the same final state (same shared memory, same computation graph).

► **Theorem 11** (Determinism). *Let P be determinacy-race-free. If $P \Downarrow \sigma$ and $P \Downarrow \sigma'$, then $\sigma = \sigma'$.*

Proof. By Lemma 10, we have σ_c such that $\sigma \rightarrow^* \sigma_c$ and $\sigma' \rightarrow^* \sigma_c$. Given that neither σ nor σ' can proceed, we must have $\sigma = \sigma' = \sigma_c$. ◀

4 DRDP Race Detection Algorithm

Race detection for parallel programs has evolved with the development of parallel programming models. The widely-used ThreadSanitizer [33] and other vector-clock-based race detectors [16, 21] work well for multithreaded programs with lock-based synchronization. More recently, task-based parallel programming models have gained popularity for developing parallel programs intended to be determinate, i.e., these programs are always expected to compute the same results when given the same inputs. The work to be carried out is decomposed into a large number of user-defined fine-grained tasks, and dependencies among tasks are specified using join operations/futures/promises rather than locks. Task-parallel programs execute on a group of *worker* threads, with the actual schedule of tasks on worker threads determined adaptively and automatically by a runtime system. Although vector-clock-based race detectors can be applied to task-parallel programs at the worker-thread level, such an approach may also exhibit false negative results. For two tasks executing on the same worker thread, a vector-clock-based race detector may enforce a happens-before relation between them, and then fail to identify potential data races³. On the other hand, it is not practical to use such race detectors by treating each task as a thread. Task-parallel programs may create millions of tasks at runtime, making it intractable to store associated vector clocks of spawned tasks in the memory space. Other researchers have also made similar observations about the limitations of using the vector clock approach for task parallelism [29, 44, 45].

Per-input dynamic race detectors designed for task parallelism can usually be classified by the task-parallel constructs they support. Different task-parallel constructs impose different structural constraints on the computation graphs generated by programs, and the determinacy race detection problem becomes more challenging as the computation

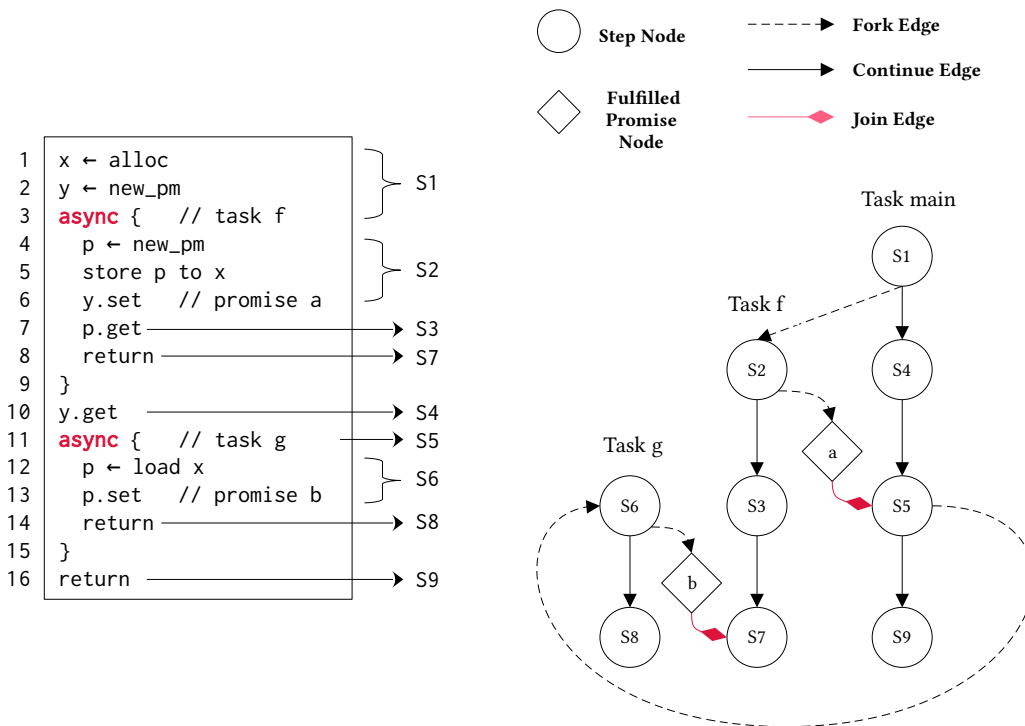
³ ThreadSanitizer limits the vector clock size to 256 [11], and can also exhibit false negatives for programs with larger numbers of threads.

13:12 Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises

graphs become more general [4, 15, 26, 30, 31]. For example, SP-Bags is a race detection algorithm designed for spawn-sync task-parallel programs which only generate fully strict computation graphs [15]. ESP-Bags is an extension of SP-Bags that can support the more general terminally strict computation graphs [30] generated by async-finish task parallelism. More recent algorithms have been introduced for task-parallel programs with futures, one as an extension to async-finish constructs [35] and others as an extension to spawn-sync constructs [37, 43].

In this section, we introduce DRDP, a dynamic determinacy race detector taking task parallelism and promises into account. DRDP is based on our theoretical conclusion in Section 3. For better time and memory efficiency, we make two revisions to the notation of computation graph. Such changes do not affect the precision of race detection.

- We introduce *step nodes* to replace task nodes. A step node is a sequence of statements without task creation, `set`, or `get` except the last statement. For example, node *s1* in Figure 6 is a step node ending with a task creation.
- We simplify the computation graph construction related to promises. Every promise has only one corresponding node in the computation graph (see promises *a* and *b* in Figure 6), created when the `set` happens.



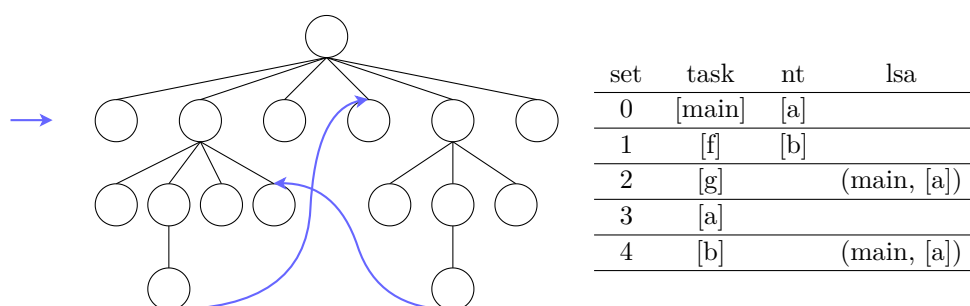
■ **Figure 6** Example program and its computation graph.

As with other dynamic race detectors, DRDP consists of 1). a *reachability data structure* that keeps track of the happens-before relationship and 2). a *shadow memory* that records access history for every memory location. We first introduce our reachability structure, which is built on-the-fly as the program executes.

4.1 DRDP Reachability Data Structures

The reachability data structure of DRDP is adapted from previous work [31, 35] on async-finish task parallelism (but no support for promise). The statement “finish { s }” causes the current task to execute the body s and wait for all spawned tasks, including transitively created tasks within the finish statement.

We leverage *Dynamic Program Structure Tree* (DPST) [31] to encode happens-before relations created by async-finish parallelism efficiently. Figure 7 shows an instance of DPST associated with the example program in Figure 6. DPST resembles the computation graph by nature. Each leaf node in the DPST represents a step node. Each internal node in the DPST is either an `async` or a `finish` node, denoting an instance of such construct in the program. Node $A1$ in Figure 7 represents the `async` in line 3 of the example program and $A3$ represents the one in line 11. The other two `async` nodes, $A2$ and $A4$, represent the `promise set` in lines 6 and 13.



■ **Figure 7** DPST and set information for the example program.

For any two step nodes s_i, s_j in a DPST, their happens-before relation (\rightsquigarrow) can be determined by examining the children of their least common ancestor (LCA). Assume s_i is to the left of s_j in the DPST. Among all children of the two node’s LCA (denoted as $lca(s_i, s_j)$), a node V (denoted as $lca_lc(s_i, s_j)$) must exist such that V is a ’s ancestor or a itself. If V is not an `async` node, $a \rightsquigarrow b$; otherwise, $a \parallel b$.

As an example, in Figure 7, step nodes $s1$ and $s2$ have the root R as their LCA. The node V , in this case, is $s1$ itself because $s1$ is a child of R . Since $s1$ is a step node, DPST will report $s1 \rightsquigarrow s2$, which is confirmed by the path from $s1$ to $s2$ in Figure 6.

Limitation of DPST. DPST may report incorrect happens-before relationships when naively applied to programs with promises. Let us consider step nodes $s2$ and $s6$ in Figure 7. Their LCA is R , and among R ’s children, the one that is $s2$ ’s ancestor is $A1$. The DPST-based happens-before check will decide that $s2 \parallel s6$. However, Figure 6 clearly shows a path from $s2$ to $s6$, so the happens-before check returns an incorrect result. The reason is that DPST does not consider synchronization semantics brought by promises.

Thus, we need to maintain additional information for those happens-before relations incurred by promises. We refer to those promise join edges in a computation graph as *non-tree joins* (nt) because DPST does not store them. For other task joins kept by DPST, we call them *tree joins*. The problem turns into how to store these non-tree joins efficiently.

Inspired by previous work [35], we use disjoint sets [12] to effectively save non-tree joins information. Tasks synchronized by tree joins will be grouped into the same set. Each set will maintain its non-tree joins, plus the lowest ancestor with at least one non-tree join, which we refer to as the *least-significant ancestor* (LSA). How do we use and maintain set, non-tree joins, and LSAs will be introduced in Section 4.3.

DRDP Example. From a high-level perspective, DRDP checks happens-before relations in two stages. It first carries out the DPST-based happens-before check. If the happens-before check returns that the two steps may happen in parallel, DRDP conducts an additional graph traversal on the computation graph. The graph traversal loops through non-tree joins and LSAs to see if the two steps are ordered by non-tree joins.

We now explain how DRDP can find out $s2 \rightsquigarrow s6$ by the information in Figure 7. DRDP first checks DPST and finds $lca_lc(s2, s6)$ is an `async` node, so it continues with the graph traversal. Because $s6$ is in task g and task g is in set 2, DRDP examines the column `nt` of set 2. In this case, we do not have any non-tree join for set 2. Finally, DRDP checks the column `lsa` of set 2, realizes set 2's `lsa` is the main task, and the main task has a non-tree join from task a . Now, DRDP knows $a \rightsquigarrow s6$. The goal becomes deciding if $s2 \rightsquigarrow a$. DRDP inspects DPST and finds $lca_lc(s2, a)$ is a step node, which indicates $s2 \rightsquigarrow a$. Now DRDP can conclude $s2 \rightsquigarrow s6$.

4.2 DRDP Task Scheduling and Shadow Memory

Existing race detectors [4, 15, 30, 35, 37] that execute programs sequentially usually rely on the *serial-projection property*, which ensures that there is a legal scheduling strategy that executes a parallel program sequentially without any blocking. Crucially, however, programs with promises do not enjoy the serial-projection property.

As a solution, our runtime follows a depth-first execution order and switches to task T 's parent task recursively when T is blocked. As soon as some other task S sets the promise that T is waiting on, the worker thread will suspend task S and resume the execution of task T . If the fulfilled promise P enables more than one task, we execute these enabled tasks in the same order as they were placed in the waiting queue for promise P . The worker switches back to task S after all tasks previously blocked on promise P are finished. If a deadlock exists in the program, our race detector works up to the point of deadlock.

Shadow memory also needs careful design due to possible blockings. For each memory location, we save the last step node that writes to it, plus a read list that records all step nodes reading from the memory location after the last write. When a write occurs, if no race is found with the recorded historical accesses, we empty the reader list and save the current step node as the last write. When a read occurs, if no race is found with the last write, we add it to the reader list.

4.3 Algorithm

DRDP algorithm is presented in Figure 8 and Figure 9. Figure 8 explains how we maintain the reachability structure and the shadow memory, which happens per instruction of the program being executed; Figure 9 defines procedure *PRECEDE* for happens-before check. When encountering a read or write to memory location M , DRDP will do race checks as in lines 40 - 44 and lines 45 - 55; if the function reports a race, it means a race exists between current access and previous access to M .

The built-in function $run_eager(U)$ suspends the execution of current task T and starts executing task U . For tasks returning from blocking, run_eager will resume their execution from the first statement after blocking.

Now we briefly explain each callback executed by DRDP.

Task creation: When a task is created, we initialize some information and create a disjoint set. The worker then starts executing the task until finished or blocked.

Task termination: We set the task state from active to finished.

<pre> // Task creation: // task T creates task U 1 S_U = new set_info(U) 2 set[U.id] = S_U 3 U.state = ACTIVE 4 U.parent = T.id 5 6 S_T = set[T.id] 7 if S_T.nt.isEmpty() then 8 S_U.lsa = S_T.lsa 9 else 10 S_U.lsa = lsa_info(T.id, S_T.nt) 11 end 12 run_eager(U) </pre> <hr/> <pre> // Task termination: // task T terminates 13 T.state = FINISHED </pre> <hr/> <pre> // Promise set: // task T sets promise P 14 P.setter_task_id = T.id 15 P.empty_task_id = (new task()).id 16 foreach task X waiting on P do 17 run_eager(X) 18 end </pre> <hr/> <pre> // Promise get: // task T gets promise P 19 if P.satisfied == false then 20 T.state = BLOCKED 21 run_eager(T.parent) 22 end 23 x = new nt_info(P.empty_task_id) 24 S_T = set[T.id] 25 S_T.nt = S_T.nt ∪ x </pre>	<pre> // Finish end: // Finish F ends in task T 26 S_T = set[T.id] 27 foreach task X in F do 28 S_X = set[X.id] 30 nt = S_T.nt ∪ S_X.nt 32 lsa = S_T.lsa 34 S_T = Union(S_T, S_X) 36 S_T.nt = nt 38 S_T.lsa = lsa 39 end </pre> <hr/> <pre> // Read check: // step s reads from memory_location M 40 w = M.writer 41 if PRECEDE(w, s) == false then 42 report race 43 end 44 M.reader_list = M.reader_list + s </pre> <hr/> <pre> // Write check: // step s writes to memory_location M 45 w = M.writer 46 if PRECEDE(w, s) == false then 47 report race 48 end 49 foreach r in M.reader_list do 50 if PRECEDE(r, s) == false then 51 report race 52 end 53 end 54 M.reader_list = {} 55 M.writer = s </pre>
--	--

■ **Figure 8** DRDP algorithm parts that maintain data structures and shadow memory.

Promise set: We create an empty task, and if any task is enabled, the worker will begin to execute enabled tasks. The worker will go back to the current task sometime in the future.

Promise get: If the promise is not yet set, we block the current task and switch to its parent recursively; if the promise is set, we add its empty task to the current set's nt.

Finish end: This happens at the end of a finish statement. The finish's owner task will merge all spawned tasks in the finish by keeping its original LSA plus unionizing the tasks and non-tree joins (*nt*) from the merged sets.

Read check: If current step *s* reads from memory location *M*, we first check race against *M*'s *writer*. If no race is reported, we directly add *s* to the *reader_list*.

Write check: If current step *s* writes to memory location *M*, we check race against *M*'s *writer* and all recorded steps in *M*'s *reader_list*. If no race is reported, we clear the *reader_list* and update *M*'s *writer* to be the current step.

Reachability query: Figure 9 elucidates how we perform reachability checks in DRDP. Given two step nodes *a, b*, we first examine if *a* precedes *b* by inspecting DPST. If *lca_lc(a, b)* is not an **async** node, we return true; this is reflected in lines 3-5. Next, we check if the previous task is still active starting from line 7. If *a*'s task is still active, either *b* is *a*'s

descendant, or a 's task sets the promise that b depends on; both cases indicate $a \rightsquigarrow b$. The remaining code in Figure 9 conducts a breadth-first search on the computation graph based on fields nt and lsa . Lines 23-39 search through non-tree joins. Lines 41 to 57 search through lsa 's non-tree joins. We return false if there are no more step nodes to search.

► **Theorem 12.** *DRDP's race detection algorithm is sound and precise. For a program P , if no execution with input ψ has any determinacy race, DRDP will not report any race (sound). For a program P , if any execution with input ψ has a determinacy race, DRDP will report the race (precise).*

Proof. To relate the theoretical result given by Theorem 11, the input ψ here refers to the initial program state $\text{root}(P)$. The proof is presented in Appendix A. ◀

► **Theorem 13.** *Given a parallel program consisting of $async$, $finish$ and $promise$ that runs in time T_1 on one worker. Assume it creates P promises and Q $async$ tasks. Let H be the height of DPST at the end of the execution, and m be the number of non-tree joins. DRDP can be implemented to check this program for determinacy races in $O(T_1 * Q * H * m * \alpha(T_1, P + Q))$ time, where α is the inverse Ackermann function.*

Proof. For a single run of PRECEDE, it could take up to $O(H * m * \alpha(T_1, P + Q))$ in the worst case if it checks all non-tree joins, and each check contains one DPST traversal plus a disjoint set operation. The maximum count of disjoint sets is $(P + Q)$ because we create one set for each task and one set for each empty task.

The PRECEDE routine may be called Q times for each shadow memory location access. This is because the reader list for a single memory location can be as large as size Q if we save all the tasks. We have at most T_1 shared memory access; thus, with DRDP checking races, the original program can be finished in $O(T_1 * Q * H * m * \alpha(T_1, P + Q))$ time. ◀

4.4 Optimizations

We introduce two optimization techniques used for DRDP. The impacts of these optimizations are respectively evaluated in Sections 5.5 and 5.6.

4.4.1 Adaptive Selection of Graph Traversal Order

A complete computation graph traversal can become necessary without restricted graph structures that enable fast encoding and checking reachability. To accelerate this part, how to traverse all non-tree joins and lsa is the key. We optimize the graph traversal in two ways: first, rather than conducting a depth-first search starting from the current node, we apply a breadth-first search instead; second, when iterating through non-tree joins, we start from the latest join to the oldest. The impact of the proper selection of these two choices is evaluated in Section 5.5.

4.4.2 Redundant Check Elimination

A single step node may access the same memory location multiple times. This may introduce a substantial amount of unnecessary duplicate checks. We present the performance improvement by skipping these redundant checks in Section 5.6. Here we introduce our approach based on the polyhedral model, a powerful linear algebraic framework for affine program analysis, transformations, and code generation.

```

Input: step nodes a, b
1 Procedure PRECEDE(a, b)
2    $S_B = \text{set}[b.task\_id]$ 
3   if  $lca\_lc(a, b).type \neq ASYNC$  then
4     | return true
5   end
6
7   if  $task[a.task\_id].state == ACTIVE$  then
8     | return true
9   end
10
11  // breadth-first search the computation graph via nt and lsa
12   $visited = \text{set}()$ 
13   $nt - steps = \text{deque}()$ 
14   $lsa - sets = \text{deque}()$ 
15  foreach  $t$  in  $S_B.nt$  do
16    |  $nt - steps.push\_back(task[t.task\_id].last\_step\_node)$ 
17  end
18
19  if  $S_B.lsa \neq NULL$  then
20    |  $lsa - sets.push\_back(S_B.lsa)$ 
21  end
22
23  while true do
24    | while  $nt - steps.size > 0$  do
25      |  $step = nt - steps.pop\_front()$ 
26      | if  $lca\_lc(a, step).type \neq ASYNC$  then
27        | return true
28      | end
29
30      |  $visited.insert(step.task\_id)$ 
31      |  $S_{step} = \text{set}[step.task\_id]$ 
32      | foreach  $t$  in  $S_{step}.nt$  do
33        | | if  $t.task\_id$  not in  $visited$  then
34          | | |  $nt - steps.push\_back(task[t.task\_id].last\_step\_node)$ 
35          | | |  $visited.insert(t.task\_id)$ 
36        | | end
37      | end
38
39      |  $add\ S_{step}.lsa\ \text{to}\ lsa - sets\ \text{if\ exists}$ 
40    | end
41
42    | while  $lsa - sets.size > 0$  do
43      |  $lsa = lsa - sets.pop\_front()$ 
44      |  $S_{lsa} = \text{set}[lsa.task\_id]$ 
45      |  $add\ S_{lsa}.lsa\ \text{to}\ lsa - sets\ \text{if\ exists}$ 
46      | foreach  $t$  in  $lsa.nts$  do
47        | |  $taskt = task[t.task\_id]$ 
48        | | if  $taskt.id$  in  $visited$  then
49          | | | Continue
50        | | end
51        | | if  $lca\_lc(a, taskt.last\_step\_node) \neq ASYNC$  then
52          | | | return true
53        | | end
54        | |  $add\ set[taskt.id].nt\ \text{to}\ nt - steps$ 
55        | |  $add\ set[taskt.id].lsa\ \text{to}\ lsa - sets\ \text{if\ exists}$ 
56        | |  $visited.insert(taskt.id)$ 
57      | end
58    | end
59
60    | if  $nt - steps.size == 0$  then
61      | return false
62    | end
63  end

```

■ **Figure 9** Reachability Check.

<pre> 1 /* Input code */ 2 for(int i = 0; i < n; i++) 3 for(int j = 0; j < m; j++) 4 for(int k = 0; k < l; k++) 5 S: C[i, j] += A[i, k] * B[k, j]; </pre>	<pre> 1 /* Code to scan written elements */ 2 /* 3 if (l >= 1) 4 for (int c1 = 0; c1 < n; c1 += 1) 5 for (int c2 = 0; c2 < m; c2 += 1) 6 write(C[c1, c2]); </pre>	<pre> 1 /* Code to scan read elements */ 2 if (l >= 1) 3 for (int c1 = 0; c1 < n; c1 += 1) 4 for (int c2 = 0; c2 < m; c2 += 1) 5 read(C[c1, c2]); 6 if (m >= 1) 7 for (int c1 = 0; c1 < n; c1 += 1) 8 for (int c2 = 0; c2 < l; c2 += 1) 9 read(A[c1, c2]); 10 if (n >= 1) 11 for (int c1 = 0; c1 < l; c1 += 1) 12 for (int c2 = 0; c2 < m; c2 += 1) 13 read(B[c1, c2]); </pre>
--	--	--

■ **Figure 10** Matmul input (left), loops to scan written elements (center), and read elements (right).

When the code region of interest are composed of *affine loops* – i.e., their loop bounds and array accesses are affine combinations of symbolic constants and outer loop iterators, that region is converted into SCoP format [3]. This format precisely specifies the set of read/written elements in the region via affine mapping representation. As an example shown in Figure 10, the SCoP representation of matmul after delinearization [18] is:

$$\begin{aligned}
 \text{Domain}_S &= \{S(i, j, k) \mid 0 \leq i < n \wedge 0 \leq j < m \wedge 0 \leq k < l\} \\
 \text{Write}_S &= \{S(i, j, k) \rightarrow C[i, j]\} \\
 \text{Read}_S &= \{S(i, j, k) \rightarrow C[i, j]; S(i, j, k) \rightarrow A[i, k]; S(i, j, k) \rightarrow B[k, j]\}
 \end{aligned}$$

Domain_S is the iteration space of statement S while Write_S and Read_S are respectively the mappings from statement instance $S(i, j, k)$ to the written and read elements of arrays A, B, C. The set of elements that are written/read by statement S is computed as the projection of Domain_S via $\text{Write}_S/\text{Read}_S$ mapping.

$$\begin{aligned}
 \text{Write}_S(\text{Domain}_S) &= \{C[i, j] \mid 0 \leq i < n \wedge 0 \leq j < m\} \\
 \text{Read}_S(\text{Domain}_S) &= \{C[i, j] \mid 0 \leq i < n \wedge 0 \leq j < m; A[i, k] \mid 0 \leq i < n \wedge 0 \leq k < l; \\
 &\quad B[k, j] \mid 0 \leq k < l \wedge 0 \leq j < m\}
 \end{aligned}$$

Using the above written & read element sets, as with the abstract memory layout [34] as scanning order, the loop nests that scan all the written and read elements are generated by the polyhedral code generation method [2].

$$\begin{aligned}
 \text{Layout} &= \{C[c_1, c_2] \rightarrow (0, c_1, c_2); A[c_1, c_2] \rightarrow (1, c_1, c_2); B[c_1, c_2] \rightarrow (2, c_1, c_2); \} \\
 \text{Code}_{\text{write}} &= \text{codegen}(\text{Layout} \cdot \text{Write}_S(\text{Domain}_S)) \\
 \text{Code}_{\text{read}} &= \text{codegen}(\text{Layout} \cdot \text{Read}_S(\text{Domain}_S))
 \end{aligned}$$

This polyhedral optimization phase has been implemented as a source-to-source transformation tool using PET [40] and ISL [39], integrated in the overall LLVM-based instrumentation pass (Section 5.1). The LLVM transformation pass first identifies the SCoP-convertible code regions and outputs them as sequential C code with SCoP annotations. Given SCoP region, the polyhedral phase computes the exact sets of read/written array elements and generates the loops that scan all elements only once. Finally, the output scanning loops are fed back to the LLVM instrumentation as the optimized code after array-based redundant check elimination. The code fragments generated by our polyhedral phase are shown in Figure 10 (center and right).

5 Evaluation

In this section, we evaluate a prototype implementation of the DRDP algorithm to address the following research questions:

1. Correctness (Section 5.2). How does DRDP compare with state-of-the-art data race detectors with respect to false positives and false negatives?
2. Performance (Section 5.4). How does DRDP perform in practice, and how does its time and space performance depend on dynamic characteristics of task-parallel programs with promises (DPST height, number of reads/writes, number of join operations, number of tasks created)?
3. What is the impact of graph traversal order on performance? (Section 5.5).
4. What is the impact of redundant check elimination on performance? (Section 5.6).

5.1 DRDP Implementation

We have implemented DRDP in a prototype race detector for task-parallel programs written in Habanero-C/C++ Library (HCLIB). Notice that any parallel program with promises that exhibits a determinacy race in the HCLIB version will also exhibit the race if rewritten with C++ promises. Our prototype can be downloaded here⁴, which includes

- a) an LLVM transform pass for instrumentation, and
- b) a C++ library for dynamic analysis.

The instrumentation pass is executed along with LLVM. When compiling a task-parallel program using the Clang/LLVM compiler, the instrumentation pass will inject a call into the dynamic analysis library after each read and write operation. The library also adds hooks into runtime to capture the invocations of HCLIB constructs (async, finish, promise). The library applies a direct-mapping shadow memory implementation [46]. A contiguous memory region shadows the entire address space, and each memory location’s shadow memory cell can be efficiently located using pointer arithmetic.

5.2 Correctness Evaluation

DataRaceBench [24] is a micro-benchmark suite designed to gauge the effectiveness of OpenMP data race detectors. In the latest 1.4.0 version, there are 181 C/C++ micro-benchmarks that cover the majority of OpenMP constructs. We leveraged DataRaceBench to conduct a correctness evaluation for DRDP. We found that micro-benchmarks using OpenMP tasking constructs can be transformed into equivalent HCLIB programs, and task dependencies specified by the `depend` clause can be achieved using promises. Therefore, we picked up all C/C++ micro-benchmarks containing the `depend` clause except DRB135 and DRB136. These two micro-benchmarks combine mutexes with the `depend` clause, which go beyond the set of programs captured by TP3 and also do not satisfy the determinacy property of TP3.

The evaluation results for these micro-benchmarks are shown in Table 1. The “yes/no” suffix in the benchmark name indicates whether the benchmark has a known data race. For short, we refer to these two groups of benchmarks as *yes-benchmarks* and *no-benchmarks*. The evaluation results are described using four terms in Table 1. FP and FN stand for *false positive* and *false negative*, respectively. A false-positive result means the race detector

⁴ <https://github.com/FeiyangJin/hclib/tree/ecoop>

reports false alarms on a no-benchmark. A false-negative result means the race detector misses potential races on a yes-benchmark. The other two terms, TP and TN, stand for *true positive* and *true negative*. TP/TN indicates that the race detector generates the expected result on a yes-benchmark/no-benchmark.

Evaluation results are shown for five state-of-the-art data-race detection tools, followed by our work (DRDP). The results for the five other tools were downloaded from the DataRaceBench GitHub repository [25] and evaluated on the OpenMP versions of the micro-benchmarks; these results were obtained in 2021 by the authors of DataRaceBench. All five tools were evaluated with the number of worker threads set to 8. Among the five tools, Intel Inspector [21], ThreadSanitizer [33], and ROMP [19] are dynamic race detectors, while Coderrect [36] and LLOV [5] are static race detectors. The DRDP results were obtained by converting the OpenMP benchmarks to HCLIB task-parallel programs with promises. From the results, we observe that DRDP is the only tool that does not report any false-positive or false-negative results for these benchmarks.

We were unable to identify the root cause of false-positive and false-negative results for ROMP and Intel Inspector since their papers did not provide sufficient information on how their dynamic analysis supports the `depend` clause. For ThreadSanitizer, the documentation states that it applies a fixed-size shadow cell to each memory location. When the shadow cell is full, ThreadSanitizer will randomly discard a recorded memory access to reserve space for the latest one. As a result, ThreadSanitizer may miss data races if one of the involved memory accesses has been discarded from the shadow cell thereby leading to false negatives. For Coderrect and LLOV, it appears that their support for tasking constructs and the `depend` clause is still under development which may explain the false-negative results. We would also expect false-positive results from these static analysis tools, when evaluated on larger benchmarks.

Apart from these converted micro-benchmarks from DataRaceBench, we also wrote some additional tests to check DRDP’s implementation. All converted micro-benchmarks and additional tests are included in our code repository.

■ **Table 1** Correctness evaluation results on DataRaceBench.

Benchmarks	Has Race?	Intel Inspector	ThreadSanitizer	ROMP	Coderrect	LLOV	DRDP
DRB027-taskdependmissing-orig-yes.c	Yes	TP	TP	TP	TP	FN	TP
DRB072-taskdep1-orig-no.c	No	TN	TN	TN	TN	TN	TN
DRB078-taskdep2-orig-no.c	No	TN	TN	TN	TN	TN	TN
DRB079-taskdep3-orig-no.c	No	TN	TN	TN	TN	TN	TN
DRB131-taskdep4-orig-omp45-yes.c	Yes	TP	FN	FN	TP	FN	TP
DRB132-taskdep4-orig-omp45-no.c	No	FP	TN	TN	TN	TN	TN
DRB133-taskdep5-orig-omp45-no.c	No	FP	TN	TN	TN	TN	TN
DRB134-taskdep5-orig-omp45-yes.c	Yes	TP	TP	FN	FN	FN	TP
DRB173-non-sibling-taskdep-yes.c	Yes	TP	FN	TP	FN	FN	TP
DRB174-non-sibling-taskdep-no.c	No	TN	TN	FP	TN	TN	TN
DRB175-non-sibling-taskdep2-yes.c	Yes	TP	TP	FN	FN	FN	TP
DRB176-fib-taskdep-no.c	No	FP	TN	TN	TN	TN	TN
DRB177-fib-taskdep-yes.c	Yes	TP	TP	FN	FN	FN	TP

5.3 Performance Evaluation Benchmarks and Setup

Since we could not easily locate an existing set of performance benchmarks for task-parallel programs using promises, we assembled a suite of seven benchmarks from other benchmark sets as follows. We did not convert the program with the largest lines of code for each set.

⁴ <https://github.com/LLNL/dataracebench/wiki/Tool-Evaluation-Dashboard>

We first converted the future-based `matmul`, `sort` and `strassen` (shared by Kastors) programs from [43] (originally from the Rodinia suite [9]) to use promises. Next, we examined the Kastors benchmark suite [41] for OpenMP task dependencies and converted two (`sparselu`, `poisson`) to use promises instead to implement the same dependencies. Finally, we converted two task-parallel OpenMP programs from the BOTS benchmark suite (`health`, `knapsack`) [14]. We only convert two because race detection on pure task-parallel programs has been widely studied before.

The summary of each benchmark is as follows:

- **matmul**: multiplies two matrices of size $2048 * 2048$ with base case size $64 * 64$.
- **sort**: sorts an array of size 10000000.
- **strassen**: multiplies two matrices of size $2048 * 2048$ by Strassen algorithm.
- **poisson**: solves the Poisson equation (aka jacobi iteration) on the unit square. The parameters we have are 8192,128,3 for matrix size, block size and number of iterations.
- **sparselu**: computes the LU decomposition of a sparse matrix. The parameters we have are 128 and 32 for matrix size and submatrix size.
- **knapsack**: calculates the solution of the knapsack problem with 40 items as input.
- **health**: simulates the Colombian health care system. We estimate the running time for the small model input file given in the source.

The evaluation was conducted on a single-node AMD server machine consisting of a 12-core Ryzen9 3900X running at 3.8GHZ with 128GB memory. All benchmarks were compiled using `-O3` optimizations by Clang/LLVM 14.0.0 running on Ubuntu 18.04.05. We report each benchmark’s mean execution time and memory usage of 5 runs for *base* and *rd* configurations. “base” is the program running time without race detection; “rd” is the one with full race detection. The standard deviations for both configurations are within 5%.

5.4 Performance Evaluation Results

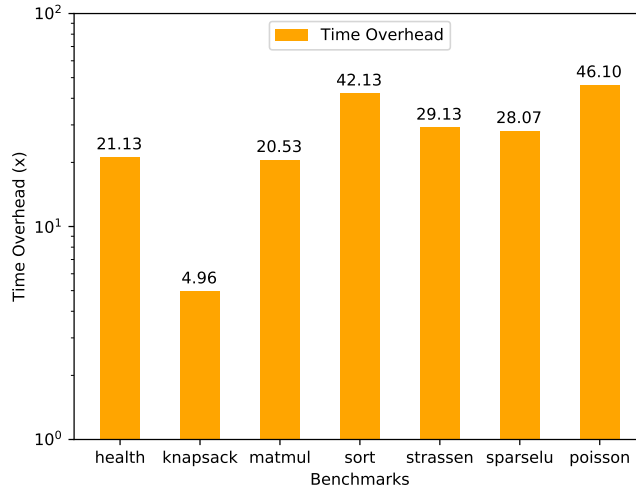
The results of our evaluation are shown in Table 2; the corresponding time overheads are in Figure 11. The first four columns show the running time in seconds and memory usage in GB, for the two configurations mentioned above. The next column “H” shows the height of DPST. Columns “Check Write” and “Check Read” are the numbers of shared memory access conducted for write and read. The following two columns, “Tree Join” and “NT Join”, are the numbers of tree and non-tree joins in the programs. Finally, column “Task” contains the number of dynamic tasks created during program execution.

■ **Table 2** DRDP performance and statistics.

Bench	Base Time	RD Time	Base Mem	RD Mem	H	Check Write	Check Read	Tree Join	NT Join	Task
health	1.51	31.99	0.45	11.64	5	49554260	84975199	2253510	0	2253511
knapsack	1.61	8.00	0.38	4.40	37	3939935	19699117	1969965	0	1969966
matmul	2.52	51.84	0.06	13.44	7	134231771	268477586	28086	0	28087
sort	0.99	40.95	0.18	4.91	15	207671935	261760850	118205	0	118206
strassen	1.37	39.97	0.20	26.12	12	123772570	243307611	29182	16380	45563
sparselu	2.46	69.01	0.04	16.70	4	187379732	362287128	12416	357760	195521
poisson	1.56	71.92	0.80	20.09	4	251658242	654802935	0	122870	106497

From the experiment statistics, we can make several observations. First, the sum of joins for `sparselu` and `poisson` surpasses the number of tasks; the reason is that each task may set and get more than one promise. This pattern is not achievable by pure task-parallel

programs. Even the future construct cannot produce more synchronization than the task number because creating a future requires creating a new task (who is responsible for setting the future). Moreover, from Table 2 and Figure 11, the race detection time overheads increase with the complexity of the parallelism pattern of the programs. Strassen, sparselu and poisson, with relatively high overheads, use promises without restrictions. The computation graphs created by the three are much more complex than the others, which is reflected by the number of non-tree joins generated.



■ **Figure 11** DRDP time overhead.

5.5 Performance Optimization: Graph Traversal Order

The motivation for selection between depth-first search (dfs) and breadth-first search (bfs) is summarized in Table 3, which shows traversal counts by different combinations of graph order (dfs vs. bfs) and iterating order through non-tree joins (back-to-front vs. front-to-back). Our optimization is extremely helpful: we always get the smallest values for columns “Max Task Visited” and “Average Task Visited” when doing bfs on the computation graph and iterating non-tree joins back-to-front. The rows with these two choices are marked in yellow in Table 3. The column “Max Task Visited” records the max number of tasks traversed in any run of the reachability check. The column “Average Task Visited” is the average number of tasks visited for each reachability check that does the graph traversal part. We picked bfs and back-to-front approach in our race detector and evaluated all benchmarks on them.

5.6 Performance Optimization: Redundant Check Elimination

Another crucial observation is that duplicate read and write checks in the same step are the bottlenecks for four benchmarks (matmul, strassen sparselu and poisson). The results in Section 5.4 included the optimized performance for these four benchmarks. In this section, we study the performance impact of the Redundant Check Elimination optimization for these four benchmarks; currently this optimization had no impact on the remaining three benchmarks evaluated in Section 5.4 because they would need interprocedural analysis across recursive calls, which is currently not performed by our optimizing compiler.

■ **Table 3** Graph traversal order comparison.

Bench	Graph Order	Input	NT Order	Min Visited Task	Max Visited Task	Average Visited Task
strassen	dfs	512 16	back to front	1	2	1.43
strassen	dfs	512 16	front to back	1	337	11.04
strassen	bfs	512 16	back to front	1	1	1.00
strassen	bfs	512 16	front to back	1	4	1.13
sparselu	dfs	32 8	back to front	1	544	8.85
sparselu	dfs	32 8	front to back	1	544	65.72
sparselu	bfs	32 8	back to front	1	2	1.07
sparselu	bfs	32 8	front to back	1	17	6.08
poisson	dfs	2048 128 3	back to front	1	32	9.54
poisson	dfs	2048 128 3	front to back	1	32	9.54
poisson	bfs	2048 128 3	back to front	1	6	2.84
poisson	bfs	2048 128 3	front to back	1	6	2.84

We record runtime statistics for these programs with or without reducing redundant checks. To measure the time overhead and memory usage, we have to re-evaluate these benchmarks on a different machine with enough memory. We present the outcomes in Table 4. The parentheses after some data show the increase/decrease percentage.

■ **Table 4** Performance comparison for reducing redundant checks.

Bench	Reduce Checks	RD Time	RD Mem	Input	Reachability Check	Write Check	Read Check
matmul	No	118.95 (+1038%)	74.77 (+3805%)	1024	1.57E+07 (+0%)	1.68E+07 (+0%)	2.15E+09 (+6299%)
matmul	Yes	10.45	1.91	1024	1.57E+07	1.68E+07	3.36E+07
strassen	No	19.73 (+233%)	12.88 (+584%)	512 16	1.94E+08 (+496%)	4.10E+06 (-47%)	7.74E+07 (+408%)
strassen	Yes	5.93	1.88	512 16	3.26E+07	7.77E+06	1.52E+07
poisson	No	214.85 (+135%)	37.87 (+129%)	7424 128 3	1.93E+09 (+84%)	2.07E+08 (-0.02%)	1.36E+09 (+154%)
poisson	Yes	91.49	16.56	7424 128 3	1.05E+09	2.07E+08	5.38E+08
sparselu	No	1137.68 (+918%)	382.37 (+2189%)	128 32	9.94E+08 (+84%)	5.86E+09 (+3029%)	1.20E+10 (+3219%)
sparselu	Yes	111.74	16.70	128 32	5.41E+08	1.87E+08	3.62E+08

All four benchmarks (matmul, strassen, poisson and sparselu) are matrix-based. As explained in Section 4.4.2, a considerable amount of redundant read checks can be eliminated in such cases. After the transformation, as shown in Table 4, the running time increases by 135% to 1038% when comparing the optimized version with the unoptimized version.

5.7 Comparison with ThreadSanitizer

Directly evaluating other tools on our benchmarks will typically generate false positives because they do not consider synchronization constraints imposed by promises. Nevertheless, given the widespread use of ThreadSanitizer in practice, we decided to evaluate it on one of our benchmarks (matmul), which is the only one for which ThreadSanitizer was able to complete successfully and that too with a smaller input. We set “report_bugs=0, ignore_uninstrumented_modules=1” for ThreadSanitizer to ensure that it ignores uninstrumented code and does not print a race report.

The issue we encountered is that ThreadSanitizer crashes on large inputs for our promise-based applications. The error messages from these crashes report a stack overflow. Most likely, it was caused by the fixed-size stack ThreadSanitizer set⁵, especially since all of our benchmarks use recursion. We use a smaller input size (128 x 128 for the matmul benchmark) instead to perform the evaluation.

⁵ https://github.com/llvm/llvm-project/blob/2e999b7dd1934a44d38c3a753460f1e5a217e9a5/compiler-rt/lib/tsan/rtl/tsan_platform_posix.cpp#L53

■ **Table 5** Performance comparison between DRDP and ThreadSanitizer.

Tool	Threads	Time Overhead	Memory Overhead	Time (ms)	Memory (MB)
ThreadSanitizer	4	20.26×	34.91×	45.8	180.49
Baseline	4	1.00×	1.00×	2.3	5.17
DRDP	1	6.10×	33.65×	17.6	166.31
Baseline	1	1.00×	1.00×	2.9	4.94

This benchmark generates 7 tasks at runtime for the given input size, so we reasonably chose to execute it with 4 worker threads for the ThreadSanitizer case. From the results in Table 5, we can see that when the input size is small DRDP has a better time performance and a similar memory performance compared with ThreadSanitizer. This is even though the ThreadSanitizer execution uses 4 threads, and the DRDP execution uses 1 thread. (The Baseline measurements were obtained on the same original version of the benchmark, but with 4 threads and 1 thread respectively.)

6 Related Work

The concept of determinacy was introduced by Karp and Miller in the late 1960s [23]. More recently, Dennis et al. reviewed this past work and introduced the concepts of structural and functional determinism [13]. Related work on the Habanero-Java programming model [7] classified task-parallel programs into seven categories where each category satisfies or does not satisfy certain properties, such as data race freedom, deadlock-freedom or determinism. The paper observed that for some parallel constructs, data race freedom implies determinacy but no proof was given for that claim. Concurrent Collection (CnC) is a dataflow-based coordination language, which was proved to be determinate [6]. Data-race free GPU programs that use barriers for synchronization have been proven to be determinate [10], though the programming model is data parallel rather than task parallel. Similarly, programs designed for heterogeneous systems can achieve portability (same input, same result regardless of the specific backend used) if data-race free [22].

There has been a long history of dynamic determinacy race detection algorithms and tools based on vector clocks [16, 21, 33]. A major advantage of the vector clock approach is that it can be applied to parallel programs with arbitrary parallel constructs, including locks and transactions (say). However, its major limitation when applied to task-parallel programs is that it can only provide guarantees on a per-schedule (rather than per-input) basis since it is not practical for vector clock sizes to be proportional to the number of active tasks.

The serial projection property has been used in past work to perform per-input race detection via sequential execution for restricted classes of task-parallel programs [4, 30]. These algorithms take advantage of the structural property of computation graphs for fork-join programs, but they do not support the arbitrary computation graphs that can be generated by task-parallel programs with promises. Surendran and Sarkar also leveraged the serial projection property to devise the first per-input dynamic race detector for task-parallel programs with futures [35]. The futureRD race detector from Utterback et al. [37] supports a restricted class of futures: it does not allow multiple `get` operations on a future handle. In contrast to previous algorithms, DRDP can support general blocking operations in task-parallel programs with promises. It also illustrates how dynamic race detection can be performed via sequential execution for task-parallel programs that do not satisfy the serial projection property.

Labeling techniques have also been used in past work on race detection for task parallelism. This approach enables reachability checking between two nodes by comparing two labels. Mellor-Crummey introduced the Offset-Span [26] algorithm as one such approach, in which the length of the label attached to each task can grow as large as the depth of nested fork structures. The SP-Bags [15] structure devised by Feng and Leiserson, and the ESP-Bags [30] introduced by Raman et al. are also examples of using labeling to record happens-before relationships.

More recently, there have been some new results on performing per-input dynamic race detection when executing programs in parallel instead of sequentially. In 2012, Raman et al. introduced the DPST data structure [31], which runs in parallel and efficiently tracks happens-before relationships of async-finish constructs. An application of DPST targets task parallelism in OpenMP has been proposed [45]. Utterback et al. [38] proposed an asymptotically optimized parallel race detection algorithm for fork-join programs. More recently, Xu et al. [43] introduced the first known parallel dynamic race detector for task-parallel programs with futures. However, none of these prior works support per-input determinacy race detection of task-parallel programs with promises.

7 Conclusions

In this paper, we addressed the problem of dynamic detection of data-races and determinacy-races in Task-Parallel Programs with Promises (TP3), which support more flexible synchronization patterns than fork-join constructs and futures. We first introduced a featherweight programming language that captures the semantics of TP3 and provides a basis for formally defining determinacy using our semantics. This definition subsumes functional determinacy (same output for same input) and structural determinacy (same computation graph for same input). We also introduced DRDP, the first-known per-input dynamic determinacy race detector algorithm for TP3, and demonstrated that it is practical to implement. To the best of our knowledge, DRDP is the first race detector that executes a task-parallel program sequentially without requiring the serial-projection property, which is a critical requirement for TP3 in general. The execution time slowdowns are all under 50×, which is comparable to overheads incurred by other dynamic race detection and debugging tools used in practice. The results also highlighted the impact of two important optimizations, traversal order and redundant check elimination, in obtaining these results. Opportunities for future work include exploring static and dynamic optimizations to further reduce the overheads in our implementation of the DRDP, as well as extensions to support determinacy race detection for promise-like constructs used in heterogeneous parallelism (e.g., CUDA graph) and in distributed-memory parallelism (e.g., MPI_Request).

References

- 1 Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H Ahn, Ignacio Laguna, Martin Schulz, Gregory L Lee, Joachim Protze, and Matthias S Müller. Archer: effectively spotting data races in large openmp applications. In *2016 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 53–62. IEEE, 2016.
- 2 Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04*, pages 7–16, USA, 2004. IEEE Computer Society.
- 3 Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In Rajiv Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2010. URL: <http://dblp.uni-trier.de/db/conf/cc/cc2010.html#BenabderrahmanePCB10>.

- 4 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1007912.1007933.
- 5 Utpal Bora, Santanu Das, Pankaj Kukreja, Saurabh Joshi, Ramakrishna Upadrasta, and Sanjay Rajopadhye. Llov: A fast static data-race checker for openmp programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26, 2020.
- 6 Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Saĝnak Taşirlar. Concurrent collections. *Scientific Programming*, 18:203–217, 2010. 3-4. doi:10.3233/SPR-2011-0305.
- 7 Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61, 2011.
- 8 Sanjay Chatterjee, Saĝnak Taşirlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with mpi. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 712–725, 2013. doi:10.1109/IPDPS.2013.78.
- 9 Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi:10.1109/IISWC.2009.5306797.
- 10 Nathan Chong, Alastair F. Donaldson, and Jeroen Ketema. A sound and complete abstraction for reasoning about parallel prefix sums. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 397–409, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535882.
- 11 LLVM Community. Tsan predefined constants for vector clocks, May 2022. URL: https://github.com/lechenyu/llvm-project/blob/main/compiler-rt/lib/tsan/rt1/tsan_defs.h.
- 12 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 13 Jack B Dennis, Guang R Gao, and Vivek Sarkar. Determinacy and repeatability of parallel program schemata. In *2012 Data-Flow Execution Models for Extreme Scale Computing*, pages 1–9. IEEE, 2012.
- 14 Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing, ICPP '09*, pages 124–131, USA, 2009. IEEE Computer Society. doi:10.1109/ICPP.2009.64.
- 15 Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258492.258493.
- 16 Cormac Flanagan and Stephen N Freund. Fasttrack: efficient and precise dynamic race detection. *ACM Sigplan Notices*, 44(6):121–133, 2009.
- 17 Standard C++ Foundation. C++11 Standard Library Extensions — Concurrency, May 2021. URL: <https://isocpp.org/wiki/faq/cpp11-library-concurrency>.
- 18 Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 351–360, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2751205.2751248.
- 19 Yizi Gu and John Mellor-Crummey. Dynamic data race detection for openmp programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 767–778. IEEE, 2018.

- 20 Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, October 1985. doi:10.1145/4472.4478.
- 21 Intel. Intel inspector, May 2021. URL: <https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html#gs.1wvmbu>.
- 22 Feiyang Jin, John Jacobson, Samuel D. Pollard, and Vivek Sarkar. Minikokkos: A calculus of portable parallelism. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 37–44, 2022. doi:10.1109/Correctness56720.2022.00010.
- 23 Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969. doi:10.1016/S0022-0000(69)80011-5.
- 24 Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17*, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3126908.3126958.
- 25 LLNL. C cpp details oct 2021, October 2021. URL: <https://github.com/LLNL/dataracebench/wiki/Tool-Evaluation-Dashboard>.
- 26 John Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91*, pages 24–33, New York, NY, USA, 1991. Association for Computing Machinery. doi:10.1145/125826.125861.
- 27 Mozilla Developer Network. Javascript reference – Promise, July 2021. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.
- 28 Oracle. Java® Platform, Standard Edition and Java Development Kit Version 17 API Specification - CompletableFuture, October 2021. URL: <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/CompletableFuture.html>.
- 29 Joachim Protze, Martin Schulz, Dong H Ahn, and Matthias S Müller. Thread-local concurrency: a technique to handle data race detection at programming model abstraction. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 144–155, 2018.
- 30 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design*, 41(3):321–347, December 2012. doi:10.1007/s10703-012-0143-7.
- 31 Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 531–542, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254127.
- 32 Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding recursive fork-join parallelism into llvm’s intermediate representation. *ACM Trans. Parallel Comput.*, 6(4), December 2019. doi:10.1145/3365655.
- 33 Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.
- 34 Jun Shirako and Vivek Sarkar. Integrating Data Layout Transformations with the Polyhedral Model. In *Proc. of IMPACT 2019*, 2019.
- 35 Rishi Surendran and Vivek Sarkar. Dynamic determinacy race detection for task parallelism with futures. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 368–385, Cham, 2016. Springer International Publishing.
- 36 Bradley Swain, Yanze Li, Peiming Liu, Ignacio Laguna, Giorgis Georgakoudis, and Jeff Huang. Ompracer: A scalable and precise static race detector for openmp programs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

- 37 Robert Utterback, Kunal Agrawal, Jeremy Fineman, and I-Ting Angelina Lee. Efficient race detection with futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPOPP '19, pages 340–354, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293883.3295732.
- 38 Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 83–94, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2935764.2935801.
- 39 Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software – ICMS 2010*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010.
- 40 Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, January 2012.
- 41 Philippe Viroulet, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. Evaluation of openmp dependent tasks with the kastors benchmark suite. In Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller, editors, *Using and Improving OpenMP for Devices, Tasks, and More*, pages 16–29, Cham, 2014. Springer International Publishing.
- 42 Caleb Voss and Vivek Sarkar. An ownership policy and deadlock detector for promises. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '21, pages 348–361, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437801.3441616.
- 43 Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. Parallel determinacy race detection for futures. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '20, pages 217–231, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3332466.3374536.
- 44 Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. Parallel data race detection for task parallel programs with locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 833–845, 2016.
- 45 Lechen Yu, Feiyang Jin, Joachim Protze, and Vivek Sarkar. Leveraging the dynamic program structure tree to detect data races in openmp programs. In *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 54–62, 2022. doi:10.1109/Correctness56720.2022.00012.
- 46 Qin Zhao, Derek Bruening, and Saman Amarasinghe. Efficient memory shadowing for 64-bit architectures. In *Proceedings of the 2010 international symposium on Memory management*, pages 93–102, 2010.

A Proof

► **Lemma 14.** *Non-tree joins (nt) and least-significant ancestor (lsa) maintain happens-before relationships correctly, given the routines in Figure 8.*

Proof. For a task T in the disjoint-set D , there are two conditions we may update D 's nt according to Figure 8.

- Lines 19 to 25: when task T gets a promise, we add the empty task created by the promise setter to D 's nt. In this way, we create dependency from step nodes (those before the set operation) in the setter task to the remaining step nodes in T .
- Line 26 to 38: when finish F ends in task T , we merge all non-tree joins from merged tasks. The merged tasks are essentially descendants of T and we keep those non-tree joins in D 's nt.

For a task T in the disjoint-set D , there are two conditions we may update D 's lsa according to Figure 8.

- Lines 1 to 12: when task T is created, if the parent has non-tree joins, we set D 's lsa to be the parent. Otherwise, we set D 's lsa to be the lsa of parent. This obeys the definition of lsa (the lowest ancestor that has non-tree joins).
- Line 26 to 38: when finish F ends in task T , we keep T 's lsa and ignore merged tasks' lsa. This is valid because merged tasks are essentially descendants of T , so for any merged task, its lsa is either T , T 's lsa or one of T 's descendants. In all cases, the nt and lsa of D already cover the information. ◀

► **Theorem 15.** *If DRDP does not report any determinacy race during an execution of program P with input ψ , then no execution of P with ψ will have a write-write race on any memory location r .*

Proof. Consider an execution δ of a program P with input ψ in which DRDP is enabled and does not report any determinacy race.

Suppose that a write-write race, χ , occurs on a memory location r in some execution δ' of P with ψ . Let W_1 and W_2 denote the two steps that write to r resulting in the race in δ' . Note that the execution δ' does not have any race until χ occurs. Without loss of generality, assume W_1 writes to r first and W_2 writes to r later in δ . We will prove by contradiction that DRDP must report the race in δ . There are two cases:

1. There are no writes to r between W_1 and W_2 in δ .

When W_1 occurs in δ' , Figure 8 lines 45 - 55 check all readers in reader list and last writer of r to see if any can execute in parallel with W_1 . Because χ is the first determinacy race in δ' , no race will be reported when W_1 occurs. We then save W_1 as the last writer to r in δ' .

When W_1 occurs in δ , because no determinacy race occurs yet, we then save W_1 as the last writer to r in δ .

When W_2 occurs in δ , we will run $PRECEDE(W_1, W_2)$ to check if $W_1 \rightsquigarrow W_2$. In δ' , $PRECEDE(W_1, W_2)$ returns false. In δ , $PRECEDE(W_1, W_2)$ returns true. We are going to show it is impossible in δ that true was returned.

In Algorithm 9, we may return true in four places: lines 4, 8, 26, and 51.

Line 4 cannot be executed in δ because DPST is the same across all executions without determinacy race. When W_2 happens, the related DPST parts that were previously generated are the same across δ, δ' . As a result, in δ , DRDP cannot return true at line 4. Line 8 cannot be executed in δ . This line only returns true if the task W_1 in is still active in δ at this point. This means either W_2 is W_1 's descendant or W_1 's task has set a promise that W_2 's task needs. Both conditions cannot be true because in δ' we have $W_1 \parallel W_2$.

Lines 26 and 51 cannot be executed in δ . We may return in these two lines if we find a path from W_1 to W_2 in a graph traversal over the computation graph. The validity is proved in Lemma 14.

2. There are writes to r by steps $W_i \dots W_j$ between W_1 and W_2 in δ .

In δ' , the happens-before relationship must be $W_1 \rightsquigarrow W_i \rightsquigarrow W_{i+1} \rightsquigarrow \dots W_{j-1} \rightsquigarrow W_j$ because χ is the first race in δ' . As $W_1 \parallel W_2$, we have $W_j \parallel W_2$, so W_j and W_2 is also a pair of write that leads to a race. Because the related computation graph parts are the same across data-race free execution, the same happens-before relationship exists in δ . When W_2 occurs in δ , the shadow memory has W_j as the last writer. We will run $PRECEDE(W_j, W_2)$ to check if $W_j \rightsquigarrow W_2$. In δ , $PRECEDE(W_j, W_2)$ cannot return true, which can be proved similarly as part 1. This is a contradiction with the statement DRDP does not report any race in δ . ◀

► **Theorem 16.** *If DRDP does not report any data race during an execution of program P with input ψ , then no execution of P with ψ will have a read-write determinacy race on any memory location r .*

Proof. Consider an execution δ of a program P with input ψ in which DRDP is enabled and does not report any determinacy race.

Suppose that a read-write race, χ , occurs on a memory location r in some execution δ' of P with ψ . Let R_1 and W_1 denote the two steps that read and write r resulting in the race in δ' . Because this is a read-write data race, R_1 occurs before W_1 in δ' . Note that the execution δ' does not have any race until χ occurs. We will prove by contradiction that DRDP must report the race in δ . There are two cases:

1. R_1 executes before W_1 in δ .
 - a. There are no writes of r between R_1 and W_1 in δ .
When R_1 occurs in δ , we check race with the last write. We then save R_1 to the reader list of r in δ .
When W_1 occurs in δ , we will run $PRECEDE(R_1, W_1)$ to check if $R_1 \rightsquigarrow W_1$. The call returns true in δ . This is impossible. The reasoning is similar to Theorem 15 part 1.
 - b. There are writes of r by steps $W_i \dots W_j$ between R_1 and W_1 in δ .
Theorem 15 states that if there is a write-write race in the program P , DRDP will always report it. This means if there exists a race in any pair of writers in $W_i \dots W_j, W_1$, DRDP must find it. Because in δ execution, DRDP does not report any write-write race, we must have $W_i \rightsquigarrow W_{i+1} \rightsquigarrow \dots \rightsquigarrow W_{j-1} \rightsquigarrow W_j \rightsquigarrow W_1$. This relationship is the same in δ' .
As a result, we can conclude in δ' , we have $R_1 \parallel W_i$, otherwise there cannot be a race between R_1 and W_1 in δ' . In δ , when W_i occurs, DRDP must report the race. The reasoning is similar to part a. This is a contradiction with the statement that DRDP does not report any race in δ .
2. W_1 executes before R_1 in δ .
 - a. There are no writes of r between W_1 and R_1 in δ .
The proof is similar to Theorem 15 part 1. We omit the details.
 - b. There are writes of r by steps $W_i \dots W_j$ between W_1 and R_1 in δ .
The proof is similar to Theorem 15 part 2. We omit the details. ◀

► **Theorem 17.** *If DRDP does not report any determinacy race during an execution of program P with input ψ , then no execution of P with ψ will have a write-read determinacy race on any memory location r .*

Proof. We omit the proof because it is similar to proof for Theorem 16. ◀

► **Theorem 18.** *If DRDP reports a determinacy race on r during an execution of program P with input ψ , then at least one execution of P with ψ will have this determinacy race on r .*

Proof. In Algorithm 9, we may return true in four places: lines 4, 8, 26, and 51. If DRDP reports a write-write race or a read-write race or a write-read race, we know none of the lines is executed. The validity of the check is explained in Theorem 15 part 1.

From the definition of determinacy race and the fact DRDP reports a race, we know at least one execution of program P will show the race. ◀

► **Theorem 19.** *The race detection algorithm described in Figure 8 and 9 is sound and precise.*

Proof. Theorem 15,16,17 show that the algorithm is sound for a given input. Theorem 18 proves that the algorithm is precise for a given input. ◀

Algebraic Replicated Data Types: Programming Secure Local-First Software

Christian Kuessner 

Technische Universität Darmstadt, Germany

Ragnar Mogk 

Technische Universität Darmstadt, Germany

Anna-Katharina Wickert 

Technische Universität Darmstadt, Germany

Mira Mezini 

hessian.AI, Darmstadt, Germany

Technische Universität Darmstadt, Germany

Abstract

This paper is about programming support for local-first applications that manage private data locally, but still synchronize data between multiple devices. Typical use cases are synchronizing settings and data, and collaboration between multiple users. Such applications must preserve the privacy and integrity of the user's data without impeding or interrupting the user's normal workflow – even when the device is offline or has a flaky network connection.

From the programming perspective, availability along with privacy and security concerns pose significant challenges, for which developers have to learn and use specialized solutions such as *conflict-free replicated data types* (CRDTs) or APIs for centralized data stores. This work relieves developers from this complexity by enabling the direct and automatic use of algebraic data types – which developers already use to express the business logic of the application – for synchronization and collaboration. Moreover, we use this approach to provide end-to-end encryption and authentication between multiple replicas (using a shared secret), that is suitable for a coordination-free setting. Overall, our approach combines all the following advantages: it (1) allows developers to design custom data types, (2) provides data privacy and integrity when using untrusted intermediaries, (3) is coordination free, (4) guarantees eventual consistency by construction (i.e., independent of developer errors), (5) does not cause indefinite growth of metadata, (6) has sufficiently efficient implementations for the local-first setting.

2012 ACM Subject Classification Information systems → Data management systems; Computer systems organization → Dependable and fault-tolerant systems and networks; Security and privacy → Cryptography

Keywords and phrases local-first, data privacy, coordination freedom, CRDTs, AEAD

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.14

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.26>

Software (Source Code): <https://github.com/rescala-lang/REScala>

archived at `swh:1:dir:9d1f296a61ad08d53d81f8e8042373e82d0a3e84`

Funding This work was funded by the German Federal Ministry of Education and Research together with the Hessen State Ministry for Higher Education (ATHENE), the German Research Foundation (DFG) within the Collaborative Research Center 1053 MAKI, the LOEWE initiative (Hesse, Germany) within the emergenCITY center, and the German Federal Ministry for Economic Affairs and Climate Action (BMWK) project SafeFBDC (01MK21002K).



© Christian Kuessner, Ragnar Mogk, Anna-Katharina Wickert, and Mira Mezini;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 14; pp. 14:1–14:33



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Today, the dominant software architecture for distributed applications is centralized. This is true for a wide variety of application types, such as single user applications deployed on multiple devices (e.g., calendars, notes, email, etc.), software that enables multi-party collaboration (e.g., shared calendars, document editors, business workflows, etc.), and software for autonomous systems with remote control and interactions (e.g., home automation and autonomous vehicles). Data is collected, managed, and processed in the cloud. Devices at the edge – owned by individuals and companies – serve merely as gateways to the cloud. Such an architecture has strengths, but also several weaknesses: It causes undue lost control over data ownership and privacy, lack of offline availability, poor latency, inefficient use of communication infrastructure, and waste of powerful computing resources on the edge.

To address these issues, local-first software design principles [25] call for “data confidentiality and privacy by default” and “ultimate ownership and control” by the user – both to be achieved by moving data storage and processing to the edge. However, developing local-first applications is challenging. Crucially, suitable existing mechanisms for efficient decentralized data management, specifically *coordination-free replicated data types (CRDTs)* [52], were invented for the geo-replicated database setting, which differs significantly from the local-first setting.

First, each local-first application has its own unique application-specific data model, designed by developers to encode domain knowledge. Developers have to figure out how to map application-specific data models to CRDTs, which are only available “off-the-shelf” in the form of databases [50] or libraries with a fixed API [24, 37]. Designing application state based on a fixed set operations is known to cause application design issues [11], because it requires translation between the application domain model and the fixed set of operations.

Second, in a local-first setting, a diverse set of networks is used for state synchronization. But general off-the-self CRDTs are designed for the geo-replicated database setting. The assumed network has mostly available direct connections between replicas, i.e., systems are designed to deal with seconds or minutes of latency between data centers. In contrast, local-first replicas (on user devices) have varied network conditions, ranging from always online devices, to personal computers that are turned off when unused, to mobile devices that only synchronize data when connected to a Wi-Fi network. In such a setting, we cannot assume direct connections between devices. Notably, this implies that connection oriented security protocols (e.g., TLS) are not applicable. A common solution for such indirect communication are cloud servers that act as intermediaries, i.e., as post offices that store, manage, and forward messages – further complicating the network model.

Third, the geo-replicated and the local-first settings have different security assumptions. Local-first applications often process personal data, with better data privacy and security being a selling point to users of local-first software. Existing security efforts for CRDTs in the geo-replicated setting [4] do not apply, due to weak attacker models and reliance on encrypted direct connections. Specifically, communication over untrusted intermediaries must not jeopardize the principles of local-first software, i.e., intermediaries must be unable to inspect or modify application data.

To fill the gaps, we propose *algebraic replicated data types (ARDTs)* – algebraic data types (ADT) that provide, by construction, provably consistent decentralized data management. ARDTs are delivered as a library that integrates seamlessly with existing language support for function composition and algebraic data types. Behind the scenes, ARDTs combine the theory of consistency as logical monotonicity (CALM theorem) [12] with delta replication [2]

for efficient and correct synchronization. Moreover, ARDTs also offer an encryption layer for efficient synchronization over untrusted intermediaries. Overall, ARDTs provide the ease of development of traditional applications, the privacy advantages of local-first, with the data sharing advantages of clouds – independent of the underlying network. We evaluate the proposed ARDT-based design of local-first applications on a case study and with micro-benchmarks. The results show that (a) typical local-first applications can be implemented with negligible performance overhead compared to existing data synchronization and UI rendering costs, and (b) encryption comes with minimal computational costs and with predictable, reasonable storage overhead on intermediaries.

In summary, our contributions are:

- A critical analysis of the state-of-the-art in developing local-first applications (Section 2).
- Use of standard ADTs suitable for application design for replication the realm of local-first software (Section 3).
- A novel synchronization-free authenticated encryption scheme, itself provided as an algebraic replicated data type (Section 4). As part of designing the encryption scheme, we contribute a systematic analysis of the suitability of existing encryption primitives for decentralized synchronization protocols (Section 5).
- An implementation of our proposal as an embedding into Scala along with a systematic empirical evaluation (Section 6).

2 State of the Art and Problem Statement

Below, we discuss two families of existing approaches, which are relevant for our work: (a) dedicated systems for collaborative workflows and (b) approaches to replicated data types employed in geo-replicated data stores. We briefly introduce relevant and missing features with respect to developing secure local-first software. We also introduce existing building blocks for distributed systems programming, which we adopt and combine to exploit their advantages in our setting.

2.1 Systems for Distributed Workflows

There are two kinds of systems for distributed workflows. The first kind has automated handling of conflicts at the price of centralized coordination; prominent examples are Google Docs or Firefox Sync. The other kind has flexible replication that does not rely on centralization; the most prominent example is Git, which allows for replication via different intermediaries including specialized ones like GitHub, or general ones like email. Similarly, systems like Syncthing and Resilio Sync enable peer-to-peer file synchronization in an arbitrary network topology, and even support encrypted intermediaries. But Git, Syncthing, and Resilio Sync require manual user intervention for conflict resolution.

We aim for combining flexible and secure data synchronization à la Git with automated conflict resolution à la Google Docs. Crucially, we aim to offer this combination to general-purpose programs with unconstrained types of data. This is unlike the above solutions, which target specific use cases and specific types of data. For example, Google Docs builds on research around operational transform [54] to enable efficient synchronization specifically for text documents. Such specialized solutions are infeasible for arbitrary local-first applications from different domains. Adapting existing solutions would require developers to become experts and understand the underlying assumptions and data models, or otherwise risk to introduce errors into an adaptation.

2.2 Replicated Data Types in Geo-replicated Data Stores

Another class of solutions that are relevant for our purposes are those developed to enable availability in geo-replicated data stores in the presence of network partitions – a scenario that bears some superficial similarity with local-first software. In particular, the solution from this context that is most relevant to the development of local-first applications are conflict-free replicated data types (CRDTs) [52]. CRDTs are data types, whose API consists of a fixed set of query and update operators, which satisfy the condition that two replicas that know of the same updates return the same result for all queries (also known as *eventual consistency*). This property is key in supporting coordination-free synchronization. CRDTs are typically built into a replicated data store with specific assumptions about the underlying network for efficiency (e.g., availability of reliable causal broadcast, trustworthiness of the involved servers).

The assumptions built into the design of off-the-shelf CRDTs limits their applicability to local-first software development. First, application developers are left with not much choice but to express their application design using the fixed APIs of existing CRDTs. A similar approach – object-relational mappings in database-centric software – is known to be a leaky abstraction, requiring frequent security relevant changes, and does not work well together with language-based tooling [11]. Second, local-first software operates in varied network scenarios for which there is no “one size fits all” solution to handle network communication. Thus, some CRDT runtimes allow developers to provide a custom message dissemination system that is specific to their needs. However, for two common network scenarios – using a cloud provider to store and forward messages, and using epidemic routing in an ad-hoc network – messages are not secure by default. Adding security burdens developers with ensuring correct and efficient encryption of messages, a task that requires expert knowledge of both the CRDT implementation and the network dissemination scheme to accomplish correctly and efficiently.

To recap, local-first applications have multiple new challenging concerns including design of the application state with replication-awareness, efficient dissemination of messages given the target network topology, and security of exchanged data, considering that messages may be stored for a long time before delivery. Each of these concerns needs both, system-level expert knowledge and application-specific insight. It is too much to ask of application developers to become experts in all of these fields. Thus, we must make expert implementations of system-level concerns such as state synchronization, message dissemination over physical networks, and security, available in a reusable and composable manner to application developers.

2.3 Building Blocks for Algebraic Replicated Data Types

Our solution builds on insights from previous research, but makes them reusable and combinable by application developers.

CALM and lattices. The first result we exploit is the *consistency as logical monotonicity* (CALM) theorem [21]. It states that consistency is possible without coordination if and only if all replicas only add to (i.e., monotonically increase) but never invalidate prior results. Existing solutions using replicated data types generally require all operations to prove (or pray for) a correctness property related to monotonicity. For example, state-based CRDTs require all operators to monotonically increase the state according to a specified order, and operation-based CRDTs require all operators to commute with one another. However, as we want to enable developers to define their own synchronization-free replicated data types

including new operations, it would be too easy for them to accidentally introduce consistency bugs (i.e., design operations that are not eventually consistent). A constructive way to enable consistency by-design, is to restrict available programming support to monotonic functions [12]. But such an approach may be too restrictive and does not integrate well into general-purpose languages.

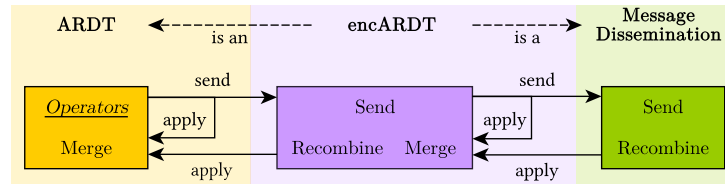
Our solution is based on join semi-lattices – a set of states (i.e., a data type) with an associative, commutative, and idempotent *merge function*. In practical terms, associativity ensures that states can be combined before transmission, commutativity tolerates disordered arrivals of messages, and idempotence handles duplicated transmissions. Any (merge) function \sqcup with the above three properties provides monotonicity in the sense above, because it implies an order on states: $s_1 \leq s_2$ if and only if $s_1 \sqcup s_2 = s_2$. Lattices have been used to reason about correctness of CRDTs since their beginning [52], but our approach makes states and merge functions directly available as building blocks for application developers.

Delta replication. We use delta replication [2] to separate efficient message dissemination from the application logic. Delta replication is a variant on state-based replicated data type design, where monotonic changes are expressed as a delta to prior state. The overall application state results by merging all deltas (according to the lattice merge). The application logic is free to generate deltas however it wants, and the message dissemination algorithm is free to optimize the transmission of deltas for the specific networking platform.

Note that not every combination of lattice semantics and message dissemination provides causal consistency (only eventual consistency). However, causal consistency always comes at the cost of waiting for messages to arrive, and most local-first applications do not require causal consistency. It is well understood how to add causal delivery to any message dissemination scheme, and doing so is compatible with our results.

But delta replication does not tell how to add encryption transparently, i.e., without having to adapt all existing message dissemination implementations (of which, each application may have its own). In general, message dissemination in a local-first setting needs to consider many application- and environment-specific interwoven concerns in addition to security. Such concerns include causal delivery, importance of messages to the application logic, and messages that become obsolete because of newer state changes. While we focus on security, our approach is designed to be parametric over the dissemination strategy for deltas, and thus enables customization.

Authenticated encryption with associated data. Local-first applications require confidentiality – the guarantee that application private data can not be accessed by unauthorized parties – and integrity – the guarantee that neither private data nor communication metadata (the associated data) can be tampered with by an attacker. These guarantees are provided by *authenticated encryption with associated data* (AEAD) [44], a family of cryptographic solutions based on symmetric-key cryptography, where only trusted parties (in our case replicas) have access to a single shared key. AEAD is well studied and widely used, e.g., in TLS 1.3 [43]. But each use of a cryptographic construction in a new field requires to carefully select concrete implementations of cryptographic functions and to ensure that they are executed with suitable parameters. The local-first setting is no exception in this respect. Specifically, in local-first applications, an unknown number of replicas need to encrypt and decrypt data using a single shared key. Widely available AEAD functions require a globally unique number (nonce) as an input for each operation using the same key. Guaranteeing global uniqueness requires coordination, which we need to avoid in the local-first setting.



■ **Figure 1** Architecture overview.

3 Algebraic Replicated Data Types

The architecture of our solution is structured in three layers depicted as colored areas in Figure 1. The layer on the left concerns algebraic replicated data types (ARDTs), which are used to model the application logic. The layer on the right concerns message dissemination over physical networks; this includes sending messages (serialized deltas) and applying a merge function to recombine received deltas into the full application state. Finally, the middle layer concerns encrypted replication (encrypted ARDTs – encARDTs). We provide a library with implementations of each layer. It includes ARDTs for common data types such as set and map, different encARDT implementations with different performance versus metadata tradeoffs, and implementations for message dissemination over TCP, Websockets, WebRTC, and disruption tolerant networks (DTNs). We expect that developers want to design new ARDTs specific for their application logic. In doing so, they can freely combine different implementations of each layer to address specific application needs. This section presents how developers design their own ARDTs and configure it to use a message dissemination module. Section 4 and Section 5 elaborate on encrypted ARDTs.

3.1 Programming and Replicating ARDTs

An ARDT is an (immutable) algebraic data type (ADT) of the host language (Scala in our case) plus a set of associated operators. The ADT values represent the (lattice) state of the ARDT. The values represent application data and, depending on the used lattice, may also include metadata for automatic merging. Operators are functions/methods that operate on an ARDT’s state. Operators may (i) just read the ARDT state to produce a value used by the application, or (ii) produce a delta that describes the desired changes to the current state. A delta is technically an instance of the ARDT, but it must first be merged into the current state to become meaningful in the context of the application.

For illustration, assume that we want to implement a local-first social media application to be used by a group of friends in a peer-to-peer network to share messages, comments, likes, and dislikes. The ARDT in Figure 2 models the state and operators of such an application¹. The `SocialMedia` type (Line 1) is defined as a product type with named components (a case class in Scala). `SocialMedia` wraps a `Map` (Scala’s built-in dictionary type) of IDs to values of type `SocialPost` (Line 6 – type parameters are in square brackets). A social post uses the built-in type `Set` for comments, two `Counters` for likes and dislikes, and a `LWW` (last-writer-wins) register for the post and comment contents. The `LWW` register is a built-in ARDT provided by our library that can be set to a new value, with the implication that all replicas will show the newest value according to a real-time clock. `Counter` is an ARDT defined in Figure 3.

¹ All code examples in the paper use Scala 3 syntax.


```

1 case class SocialMedia(sm: Map[ID, SocialPost]):
2   def like(post: ID, replica: ReplicaID): SocialMedia =
3     val increment = sm(post).likes.inc(replica)
4     SocialMedia(Map(post -> SocialPost(likes = increment)))
5
6 case class SocialPost(message: LWW[String], comments:
   Set[LWW[String]], likes: Counter, dislikes: Counter)

```

■ **Figure 2** Compositional design of the social media ARDTs.

```

7 case class Counter(c: Map[ReplicaID, Int]):
8   def value: Int = c.values.sum
9   def inc(id: ReplicaID): Counter =
10    Counter(Map(id -> (c.getOrElse(id, 0) + 1)))
11
12 object Counter: // object for static methods
13   def zero: Counter = Counter(Map.empty)

```

■ **Figure 3** The state and operators of a counter ARDT.

The operators of ARDTs implement their application logic. While `Counter` (Line 7) and `SocialMedia` (Line 1) are both wrappers around a `Map`, their operators make the difference. Each `Int` stored in the `Map` of the `Counter` ARDT represents an individual amount contributed by the specific `ReplicaID`. This is expressed by the `value` operator (Line 8). A zero counter is expressed by the empty map (Line 13). Like other immutable data structures, operators that modify ARDTs return a new state, e.g., `inc` (Line 9) increases a counter by returning a new counter. But for ARDTs it is sufficient to return a *delta* – the changed parts of the state – the rest is managed automatically by applying the merge function. For instance, `inc` (Line 9) returns only the entry with the increased values; unchanged entries in the `Map` are omitted. The `like` operator of the `SocialMedia` ARDT in Line 2, while being a bit more complex, follows the same pattern. To “like” the post with the given `ID`, it computes the increment of the likes counter (Line 3) and returns a new delta of the `SocialMedia` state, which contains only the changed `ID` and defines only the likes component² of the social post (Line 4). Returning deltas is preferable, because it is more efficient to send and merge smaller values. But since merging is idempotent, developers could also return full states without impacting behavior.

In the examples so far we assumed that a merge function for our ARDTs exist. This is indeed the case, because all built-in types we used have merge functions provided off-the-shelf by our library, and the user-defined ADTs (`SocialMedia`, `SocialPost`, and `Counter`) have their merge function automatically generated. For example, the merge functions for `Counter` and `SocialMedia` keep all entries of both maps and (recursively) merge the values that have the same key; and the merge function of the `SocialPost` merges each component individually. See Subsection 3.2 for the precise definition of these merge functions. In general, the availability of a merge function for a type `S` (e.g., `Counter` or `SocialMedia`) is modeled by the type class `Lattice[S]` below.

```

14 trait Lattice[S] { def merge(left: S, right: S): S }

```

² The syntax that looks like an assignment in Line 4 is a named parameter, and we assume that this constructor sets all other components to “empty” values (not shown in the example for brevity). The `->` operator constructs a key-value pair.

```

15 class MessageDissemination[S]:
16   def send(delta: S): Unit
17   def recombine(using Lattice[S]): S

```

■ **Figure 4** Example message dissemination module.

```

18 val smd = new MessageDissemination[SocialMedia]
19 val current: SocialMedia = smd.recombine
20 val delta: SocialMedia = current.like(myPost, replicaID)
21 smd.send(delta)
22 val updated: SocialMedia = smd.recombine

```

■ **Figure 5** Using and replicating the social media ARDT.

Specifically, we say that S is the state of an ARDT, when there exists a correct instance of `Lattice[S]`; an instance is correct if its merge function is associative, commutative, and idempotent. The correctness of the merge function is the only requirement for eventual consistency in our system, and we do not want to burden developers with its definition. Thus, our library comes with a broad range of built-in ARDTs (with state, operators, and merge function). Moreover, correct instances of `Lattice` for standard data structures, user-defined product types, and the compositions of all of the above, are automatically generated. The system produces a compilation error if no lattice instance for a custom type can be generated. In other words, eventual consistency is always guaranteed automatically.

We put ARDTs into action by composing them with a concrete message dissemination module. The API of the message dissemination module is shown in Figure 4. It allows to send and recombine a replicated state of type S (Line 15)³. The `send` method (Line 16) sends a delta message of type S . The `recombine` method (Line 17) merges all received delta messages into a full state of type S , which requires a `Lattice[S]`. The `using` keyword (Line 17) asks the compiler to provide such an instance automatically if available, or report a compilation error. Figure 5 shows how to use and replicate the social media platform. Line 18 creates a `MessageDissemination` named `smd`. We access the current state of social media (`current`) using `recombine` (Line 19). To like a post named `myPost`, we first apply the like operator on `current` to compute the delta state.⁴ Once we send delta (Line 21), the like is merged into the rest of social media, and we can access the full `updated` state by calling `recombine` (Line 22).

3.2 Lattice Composition

By design, the correctness of both the operators and the distributed consistency of ARDTs rely exclusively on their state forming a lattice, i.e., on having a correct merge function. We provide ready-to-use `Lattice` instances for a range of data types such as last-writer-wins registers, multi-version registers, and lists (RGA). We have implemented those based on existing schemes for state-based CRDTs [51, 2]. On top, our library supports automatic generation of merge functions for compound data types including associative maps, pairs, tuples, optional values, and user-defined case classes (generally all product types), given their

³ This supports multiple ARDTs by composing them into a single type S .

⁴ Note that while delta is of type `SocialMedia`, it only contains that single like.

```

23 given Lattice[Int] with
24   def merge(left: Int, right: Int): Int = max(left, right)

```

■ **Figure 6** Lattice instance for integers using their maximum.

```

25 given [A]: Lattice[Set[A]] with
26   def merge(left: Set[A], right: Set[A]): Set[A] = left union
      right
27
28 case class LWW[A](time: Time, value: A)
29 given [A]: Lattice[LWW[A]] with
30   def merge(left: LWW[A], right: LWW[A]): LWW[A] =
31     if right.time < left.time then left else right

```

■ **Figure 7** Set and last-writer-wins lattice.

constituents are ARDTs (i.e., have a lattice instance)⁵. For example, `SocialPost`'s merge is automatically generated from its constituent types, `Set`, `Counter`, `LWW`. The generation is recursive with pre-defined ARDTs (e.g., `LWW`) being the recursion anchors.

The generation for product types exploits the canonical representation of a compound data type as a function (e.g., a key-value map is a function from keys to values). The merge of two functions l and r is a new function f that merges the result of applying l and r ($f(x) = \text{merge}(l(x), r(x))$). Sum types (i.e., types representing alternatives such as colors: red, green, blue) either use built-in lattice instances, or use an explicitly specified order of the cases (e.g., $\text{red} < \text{green} < \text{blue}$) and merging returns the larger case.

In the following, we elaborate on how concrete lattice instances are defined for different ARDTs starting with the recursion anchors and ending with automatic derivation of instances for compound data types.

3.2.1 Provided and Custom Lattice Instances

We provide ready-to-use lattice instances for primitive data types and for common CRDTs. For example, the code in Figure 6 implements lattice instances for integers. The `given` keyword defines an unnamed instance of `Lattice[Int]`. The `with` keyword states that the following is the implementation of the `Lattice` methods, in this case, the implementation of `merge` as the application of the `max` function. This definition uses Scala's support for implicit values to seamlessly integrate ARDTs into the rest of the language. Methods can access an instance with the `using` keyword (as seen in the `recombine` method from Figure 4); the developer does not need to explicitly provide the instance, when the method is called.

Figure 7 shows lattice instances for sets and last-writer-wins registers. Merging sets is delegated to the existing `union` method on sets. `LWW` is a custom type, whose state associates a unique timestamp and a value. Its merge function makes an arbitrary but deterministic decision – it selects the state with the larger timestamp and ignores the other one.

We do not expect developers to define their own merge functions. It is possible to do so, by providing custom lattice instances, but carries the risk of an incorrect implementation. Instead, our library provides support to automatically derive lattice instances for custom ADTs, as elaborated in the following sections.

⁵ This is not unlike Haskell's support for deriving instances of type classes for compound data structures.

```

32 given [K, V](using Lattice[V]): Lattice[Map[K, V]] with
33   def merge(left: Map[K, V], right: Map[K, V]) =
34     left.merged(right){
35       case ((id, v1), (_, v2)) => (id, Lattice[V].merge(v1, v2))
36     }

```

■ Figure 8 Map lattice.

3.2.2 Derived Lattice Instances

Deriving lattices for a compound type makes use of lattices of its component types. Technically, this is represented as `given` instances that take other instances as parameters. In the following, we present how to derive lattices for generic map and product types. The appendix includes proofs of their correctness by showing that the respective merge functions are commutative, associative, and idempotent.

The map lattice. Figure 8 states how any `Map[K, V]` has a lattice instance, if its values `V` also have a lattice instance. Specifically, the `using` keyword states that to create the lattice instance for the map we require a `Lattice[V]` where `V` is the type of values stored in the map. The merge function (Line 33) for a map delegates to the built-in `merged` function of `Map` (Line 34). The built-in `merged` function does not automatically handle the case when a key is assigned to value in both the left and the right map and requires a custom function to handle such conflicts. We implement this function to delegate to the merge function of the value type provided by `Lattice[V]` (Line 35). We prove correctness of this merge function in Appendix A.1.

The product lattice. We support automatic generation of lattices for any product type whose elements themselves have lattice instances. In Scala, product types include tuples and case classes. Consider the exemplary case class `MyData` in the first line of the code snippet below, and an explicit definition of the automatically generated lattice instance in the second line.

```

37 case class MyData(a: A, b: B)
38 given Lattice[MyData] = Lattice.derived[MyData]

```

The `derived` method generates a lattice instance for any product type `S`. Its implementation is shown in Figure 16 (in the Appendix). At a high-level of abstraction, a lattice instance for a product is generated as follows. (i) Acquire lattice instances for each component of the product. (ii) Define the merge function for two instances of the product type (a left and a right one) to (iii) take each component of the left product and merge it with the corresponding component of the right product and (iv) return the results wrapped in a new instance of the product.

We give the full technical details of the implementation in Appendix A.2. We prove correctness of the merge function for arbitrary products in Appendix A.3.

3.3 Qualitative Assessment of Design Features

We recap key advantages and limitations of our approach to defining and using ARDTs.

Reused implementations. When designing a new ARDT, expert developers can reuse any existing data structure, as long as they can define a merge function. For instance, the `Map` data structure in our social media ARDT is a highly optimized implementation from the Scala standard library. But developers are not limited to the options in our library and are free to choose an implementation that best suits their needs. For instance, there are multiple implementation strategies for sets to choose from, including sorted trees and hash-based solutions. The decision about the particular data type to use for representing the state of custom ARDTs is decoupled from and does not affect consistency management. Our approach enables to reuse existing off-the-shelf CRDTs by providing a suitable lattice instance. State-based CRDTs have a correct merge function, which we can and have directly used for this purpose. Operation-based CRDTs can be systematically converted to state-based CRDTs [52], thus they can be reused, too.

Unified consistency management. The CALM theorem [21] implies that monotonicity is a necessary requirement for consistency without coordination. To achieve monotonicity, existing state-based CRDT implementations [52] require that operators return a state that is larger than the original one (with respect to a pre-defined order of all possible states) and operation-based CRDTs require operators to be commutative. In contrast, our approach automatically enforces monotonicity of operators by merging their delta result with the current state. Thus, application developers do not need to reason about consistency when designing operators. The potential for introducing consistency bugs is limited to custom merge functions, which we assume to be designed by experts.

To illustrate the positive effects of this, consider again the `Counter` ARDT in the social media application. We only have to ensure that operators implement the intended application semantics, but we are always guaranteed consistent results. That is, the developer may make a mistake and the increment operator does not increment the value of the counter, as it is supposed to do. But it is guaranteed that the operator exhibits the same (erroneous) behavior on all replicas. This is in contrast to classic CRDTs, where an incorrect operator may lead to different states on different replicas. Due to unified consistency, distributed correctness boils down to correctness of a single replica, i.e., we get along with local reasoning, which simplifies development and testing.

Finally, since correctness relies exclusively on the properties of the merge functions, reasoning about consistency and ensuring it automatically is greatly simplified. An indication for this are the proofs (in the Appendix) for generated merge functions presented in Subsubsection 3.2.2. First, they are of manageable size. Second and more importantly, one can prove the correctness of individual merge functions independently of other merge functions and operators, because they do not rely on any global assumptions. Correctness for all composed data types then follows automatically from the individual proofs.

Versatile message dissemination. Local-first applications may run on diverse communication infrastructures, especially when considering various potential intermediaries ranging from a centralized server, to a shared network disk, to passing data along multiple ad-hoc Wi-Fi connections, to storing messages on a USB drive and sending the latter via physical mail.⁶ Even though concrete strategies for message dissemination are not a focus of our

⁶ Networks, where messages are not exchanged directly, but rather stored and forwarded until they are eventually received, are called delay-tolerant networks (DTN) [5]). They are actively developed and researched to enable resilient communication [53, 48, 5, 6], a highly relevant area for local-first software.

contributions, our assumptions about message dissemination are explicitly designed to admit many different such strategies. Moreover, the separation of message dissemination from ARDTs enables independent improvement of separate concerns. Specifically, in Section 4, we thoroughly explore secure communication that works in any setting. On the other side of the spectrum, in Section 6, we also explore possible optimizations of message dissemination in less challenging environments such as a central server.

Limitations. According to the CALM theorem [21], coordination-free consistent replication schemes can only express algorithms that do not require consensus. This is true for ARDTs, too. Even though we support arbitrary code to express operators, the deltas produced by operators are merged back into the current state, which enforces that the actual change to the state is monotonic. For example, a decrement operation on the counter ARDT (Figure 3) could produce a delta that decrements one of the integer values in the counter. However, because merging integers (Figure 6) returns the maximum, such a delta has no effect when merged into the current value.

4 Encrypting ARDTs

The design of ARDTs is motivated by the need for a flexible encryption mechanism suited for local-first applications. In particular, encryption should be independent of the message dissemination mechanism to provide the same guarantees in any network scenario. Moreover, the encryption should enable efficient storage of encrypted data on untrusted intermediaries.

Our solution provides encryption as a special kind of ARDTs, called *encrypting ARDTs* (encARDTs in the middle of Figure 1). EncARDTs are normal ARDTs for all purposes – they can be replicated using any message dissemination mechanism, and they can be parts of composed ARDTs. EncARDTs provide encryption via their operators, specifically, they implement the message dissemination API (send and recombine) where sending encrypts and recombine decrypts the state.

For example, a naive implementation of an encARDT is shown in Figure 9. Line 41 defines the state of the encARDT as a set of encrypted values. For encryption, we rely on *authenticated encryption with associated data* (AEAD) to ensure confidentiality of the state and integrity of both the state and the metadata. There are multiple encryption primitives that provide AEAD, and we elaborate on the challenge of correct use of AEAD in a coordination-free setting in Section 5. For now, we assume that there exists a suitable AEAD module with the following interface.

```
39 def encrypt[S, A](data: S, meta: A, key: Secret): AEAD[S, A]
40 def decrypt[S, A](aead: AEAD[S, A], key: Secret): Option[(S, A)]
```

The naive encARDT in Figure 9 stores values of type AEAD. We generally refer to encrypted states as *messages* to distinguish them from the state of the encARDT itself. The send operator (Line 42) adds new messages into the encARDT, by using the `encrypt` method of the AEAD module, and producing a delta containing the message. This delta is handled as usual, i.e., it is merged into the current state using the automatically derived merge function. The recombine operator (Line 44) reconstructs the plaintext ARDT state of type `S`. To do so, all messages are processed by the `decrypt` method (Line 45), whose return value for authentication failures makes `flatMap` discard that message. The successfully decrypted messages are merged pairwise using the lattice of the plaintext ARDT `Lattice[S]`.

```

41 case class Naive[S](messages: Set[AEAD[S, Unit]]):
42   def send(data: S, key: Secret, rID: ReplicaID): Naive[S] =
43     Naive(Set(encrypt(data, (), key)))
44   def recombine(key: Secret)(using Lattice[S]): Option[S] =
45     messages.flatMap(aead => decrypt(aead, key)).map(_.data)
46     .reduceOption(Lattice[S].merge)

```

■ **Figure 9** Naive encARDT stores all states.

```

47 case class Subsuming[S](messages: Set[AEAD[S, Version]]):
48   def version: Version = messages.map(_.metadata)
49     .reduceOption(Lattice.merge[Version])
50     .getOrElse(Version.zero)
51   def send(data: S, key: Secret, replicaID: ReplicaID) =
52     val cause = version merge version.inc(replicaID)
53     Set(encrypt(recombine(key) merge data, cause, key))
54
55 given [S]: Lattice[Subsuming[S]] with
56   def merge(left: Subsuming[S], right: Subsuming[S]): Subsuming[S] =
57     val combined = left union right
58     combined.filterNot(s =>
59       combined.exists(o => s.metadata < o.metadata))
60
61 extension [S] (c: Subsuming[S])

```

■ **Figure 10** Subsuming encARDT based on version data.

Consistency of the naive encARDT directly follows from the automatic construction of the merge function, because we only ever add new messages. However, storing all messages forever is a naive solution, because the state grows indefinitely. Yet, the naive encARDT represents the realistic case where an intermediary has no further information about encrypted messages. The following sections describe how to fix the indefinite growth of required space by using associated metadata.

4.1 Pruning Subsumed States

The naive encARDT stores messages even if they are no longer relevant. As an example why this is problematic, consider the counter ARDT. The counter stores an integer per replica ID, each time a counter is incremented we store the new value and no longer need the old value for that replica ID. In such cases, we say that the old state is *subsumed* by the new state. Formally, a state s' subsumes another state s , if s' contains all updates of s , i.e., $s \sqcup s' = s'$ (where \sqcup is the merge function).

The *subsuming encARDT* attaches logical timestamps [28] in the form of *version vectors* [10] to messages as associated metadata. Version vector metadata provides an order \leq on encrypted states $e(s)$ that implies subsumption: $e(s) \leq e(s') \implies s \sqcup s' = s'$. This allows intermediaries to remove subsumed messages without inspecting their contents. Figure 10 shows the implementation of the subsuming encARDT, whose messages include the `Version` as associated metadata for the encrypted states. A version vector is semantically a counter CRDT, and `Version` uses the implementation from Figure 3, but is renamed to reflect its use within the subsuming encARDT.

```

63 case class Dotted[S](messages: Set[AEAD[S, (Dot, Set[Dot])]]):
64   def send(data: S, key: Secret, replicaID: ReplicaID) =
65     val cont = messages.flatMap(aead => decrypt(aead, key))
66     val sub = cont.filter(s =>
67       Lattice[S].merge(s.data, data) == data)
68       .flatMap(s => dotsIn(s.metadata)).toSet
69     Set(encrypt(data, (Dots.next(replicaID), sub), key))

```

■ **Figure 11** Dotted encARDT – precise subsumption.

The operators of the subsuming encARDT automatically add the correct metadata to messages. The helper method `version` (Line 48) merges all currently stored versions, thus returning the upper bound of all versions. The `send` operator increments the upper bound of versions (Line 52), implying that the new message subsumes all existing messages, but is not subsumed by any of them. To ensure this is true, the delta state to be send (`data` in Line 53) is merged with all current values in the encARDT, thus producing a full state that does contain all others. The `recombine` operator is the same as for the naive encARDT in Figure 9, hence not shown.

An explicit lattice instance implements subsumption as part of the merge function (Line 55). After computing the union of the sets of encrypted states (Line 57), the merge keeps only the states that are not subsumed by another state (Line 58); formally the kept states are $\{e(s) \mid \nexists e(s') : e(s) < e(s')\}$.

For an intuition to how a subsuming encARDT behaves, consider that a message subsumes all messages that are currently stored in the encARDT, and the merge function removes all subsumed messages. Thus, each time a replica sends a message, only that message (containing all deltas) is stored. However, when multiple untrusted intermediaries synchronize between each other, each may store multiple incomparable messages (generated by different replicas), and merging will keep all of these messages until a trusted replica decrypts and merges them.

In Appendix A.5, we prove that the subsuming encARDT is transparent, i.e., sending and recombining behaves as if we just merge states without encryption, and without removing them based on subsumption metadata. This proof includes correctness of the custom merge function (associative, commutative, idempotent).

4.2 Pruning Encrypted Deltas

With the subsuming encARDT, we lose the advantages of delta replication, because it combines all deltas into a single state when sending a message. To address the problem, *dotted encARDT* in Figure 11 store precise per-delta subsumption information in the metadata. Specifically, the metadata contains (a) a globally unique logical timestamp for the message, called a *dot* [40], and (b) the set of dots that the message subsumes. The `send` operator computes the set of messages currently contained (`cont` in Line 65) in the dotted encARDT. For each contained message, it uses the merge function (Line 67) to check if it is subsumed by the new message. Subsumption is transitive, thus the new subsumption info combines all dots in the metadata of all subsumed messages (`sub` in Line 68). Finally, the message containing only the delta (`data`) and subsumption info is returned (Line 69). The other methods (including the merge function) of the dotted encARDT are the same as for the subsuming encARDT, thus are not shown.

With dotted encARDTs, intermediaries still cannot decide if two concurrent messages from different trusted replicas subsume one another, but the messages are potentially much smaller compared to the subsuming encARDT. However, dotted encARDTs require more metadata, thus use more space when no concurrent messages occur. See Section 6 for an empirical evaluation, but keep in mind that the best choice is highly dependent on the used ARDTs and application behavior. It is possible to use different encARDTs for different ARDTs in the application, thus providing maximum flexibility. Correctness proofs for the dotted encARDTs are analogous to the subsuming encARDT, because using a more precise notion of subsumption only strengthens the preconditions of the proof.

4.3 Security Assessment

We have presented three different encARDT strategies that cover different points in the design space. Assuming that only trusted replicas know the shared secret, and that AEAD protects data confidentiality and authenticity of the contained messages, all encARDTs prevent the following attacks. Intermediaries cannot tamper with the order of data, because recombination is order independent. Replay attacks using duplicated messages also have no effect, since merging is idempotent. Intermediaries can forge new messages using incorrect keys, but these are ignored when decrypting. The only way for intermediaries to interfere is to selectively stop disseminating messages to (some or all) replicas – this is not worse than the scenario where the intermediary did not exist.

Encrypted communication may still leak information, e.g., the size of messages, and who sends which message at what time. In addition, different encARDTs have different tradeoffs. The naive encARDT leaks no metadata, but stores unneeded messages. The dotted encARDT is as precise as possible, but also leaks precise subsumption metadata. Subsuming encARDTs are in the middle. They leak the order of messages, which can be learned anyway by an attacker that can observe the overall network (a common threat model), while still enabling to remove unneeded messages.

Leaking metadata is considered as unproblematic when synchronizing rich data such as texts and images, because the contained data is not deducible by the order in which modifications happened. But it can be problematic for certain simple ARDTs, e.g., in the case of Counter (Figure 3), which has a single operation, one can deduce the current values by learning the number of messages. But these issues are not unique to our solution, and countermeasures exist [20, 56]. Moreover, because encARDTs do not require a central entity, it becomes easier to apply countermeasures. For example, one can split messages over multiple intermediaries (hence, no single intermediary may learn all metadata), or can use randomized routing such as TOR [13], because ARDTs are resilient against unreliable message delivery.

5 Coordination-free Encrypted ARDTs

The discussion in the previous section leaves out one open challenge: AEAD primitives require each call to the encrypt method to use a globally unique number (nonce). In general, ensuring global uniqueness of something requires coordination, which contradicts our goal to support coordination-free synchronization. Thus, the open question is how to guarantee global uniqueness while practically avoiding coordination. To answer this question, we are the first to analyze multiple stochastic methods of selecting unique numbers for their suitability for the local-first setting.

	Java	Web	libsodium	Tink
AES-GCM	•	•	•	•
AES-GCM-SIV				•
ChaCha20-Poly1305	•		•	•
XChaCha20-Poly1305			•	•

■ **Figure 12** Overview of supported AEAD modes in various environments.

5.1 The Study Setup

Existing solutions vary in different respects: their availability, the number of replicas that are securely supported, the number of operations that can be executed without coordination. The goal of our analysis is to delimit the chances for conflicts within common security standards.

We considered the following AEAD constructions: AES-GCM, AES-GCM-SIV, and (X)ChaCha20-Poly1305. Figure 12 shows their availability in the Java Cryptography Architecture⁷, Web Cryptography API⁸, libsodium⁹, and Tink¹⁰. All libraries support AES-GCM due to its use in the TLS specification [43]. The more modern AEAD construction ChaCha20-Poly1305 was introduced in TLS 1.3 [43] and is currently also supported by all libraries except Web Cryptography API. XChaCha20-Poly1305 [3] is an adaption of ChaCha20-Poly1305 with a larger nonce-size and proven to be at least as secure [7]; while not yet standardized by IETF, it is supported by libsodium and Tink [3]. AES-GCM-SIV [19] (implemented only in Tink) claims resistance to nonce reuse; it is also not standardized.

In summary, the best available options are:

AES-GCM Use a 64 bit random ID per replica and 32 bit replica specific counter as nonces.

Supports up to 92,000 replicas, communicating once per second for 132 years.

XChaCha20-Poly1305 Use fully random 192 bit nonces. Supports 2^{32} replicas for communicating once per millisecond for 8900 years.

Our implementation defaults to XChaCha20-Poly1305, because it allows to completely hide the use of nonces from the developer. In the following, we elaborate on how we reached the conclusion that the above solutions are the best available options.

5.2 Coordination-free Generation of Nonces for AEAD

To encrypt and authenticate a message, AEAD schemes generally require three inputs: the message, the encryption key, and a *nonce* [45]. A nonce is a **number** that must only be used **once** together with the same key. If a nonce is used multiple times, then encryption schemes leak information about the plaintext, e.g., in AES, an attacker learns the bitwise exclusive-or of messages [31]. The Nonce misuse has led to severe real-world attacks, e.g., on TLS [9] and WPA2 [57]. The issue is that the decision on how to choose nonces is left to the developer, and, unfortunately, previous research on crypto misuses has shown that developers struggle with secure choices for crypto APIs [27, 36, 41]. This is not surprising, considering that libraries like the Web Cryptography API do not even document that nonces should be unique. Ensuring uniqueness is a classical coordination problem. Thus, we discuss how to select unique nonces without coordination, while staying within generally accepted levels of certainty for the provided confidentiality.

⁷ <https://docs.oracle.com/en/java/javase/16/security/>

⁸ <https://www.w3.org/TR/WebCryptoAPI/>

⁹ <https://libsodium.org/>

¹⁰ <https://developers.google.com/tink>

5.2.1 Selecting Nonces by Space Partitioning

A textbook approach to ensure uniqueness of nonces is using a strictly monotonic counter [14]. This is the case for AEAD algorithms in TLS 1.3 [43]. Using a single counter for all replicas is not possible without coordination, since this is a prime example of mutual exclusion. An adaption of the counter approach is to partition the nonce space into multiple ranges, each exclusive to a single replica. This strategy requires coordination only once per replica. Fixed ranges are a good choice for a set of devices provided by a single authority (a single user or company). In large groups of loosely cooperating devices, however, deterministic coordination of non-overlapping nonce ranges is infeasible. An alternative approach is *cryptographically secure pseudorandom number generation* (CSPRNG).

Using replica IDs for partitioning. Certain ARDTs such as the counter (Figure 3) already require replica-specific IDs for their behavior. Therefore, it seems intuitive to reuse the replica ID to partition the space of nonces. If the chance of collisions of any two replica IDs is small enough to be negligible, this is a secure choice. In general, to ensure uniqueness, typical examples for replica-IDs are randomly generated UUIDs (as seen in automerge¹¹), or a hash of a replica-specific public-key [24], with the size of such identifiers usually being 128 bits [29]. Unfortunately, this size is too large for use with popular AEAD constructions. For example, the NIST specification for AES-GCM recommends that implementations should restrict their nonce lengths in AES-GCM to 96 bits [14]. Thus, direct use of replica IDs to partition the nonce space is not possible.

Using small random replica-specific numbers for partitioning. Instead of using the replica ID, we can generate short replica-specific numbers using a CSPRNG, but this leaves us with a probability of collisions of replica-specific numbers, thus a collision of nonces. According to the NIST specification, the probability that a nonce is reused for a given key must be less or equal to 2^{-32} [14]. Considering the birthday paradox [49], there is a surprisingly high probability that two replicas choose the same replica-specific number. With a 64-bit long replica-specific number, we can have 92,000 replicas before the collision probability reaches over 2^{-32} . Assuming 92,000 replicas are sufficient, and given the explicit 96-bit nonces of AES-GCM, a 64-bit replica-specific number leaves room for 32-bit replica-specific counters. This provides $2^{32} \approx 4.3 \times 10^9$ messages to each replica. Assuming that a replica encrypts one message per second, the counter could be used for 136 years, before requiring coordination to select a new shared secret. This is a realistic choice for local-first applications, when only AES-GCM is available.

5.2.2 Selecting Fully Random Nonces

A fully coordination-free approach to nonce generation is to rely on a CSPRNG to generate a new random nonce for each message. Literature warns against random nonces in some cases [9]. For example, nonces in TLS (using AES-GCM) consist of 32-bit part specific to the sender and connection, and a 64-bit part to ensure uniqueness [46]. With 64-bit random nonces the collision probability after encrypting $2^{28} \approx 2.7 \times 10^8$ (three hundred million) messages would be around 0.2 % and for $2^{32} \approx 4.3 \times 10^9$ messages around 39 % [9].

¹¹<https://github.com/automerger/automerger>

For using 96-bit random nonces with AES-GCM, the libsodium documentation recommends against it [30], while the documentation of Tink recommends it for “most uses” [17]. Specifically, Tink guarantees that AES-GCM with random nonces can be used for at least $2^{32} \approx 4.3 \times 10^9$ messages, while keeping the attack probability smaller than 2^{-32} [17].

This, however, is a global message limit, i.e., counting all messages encrypted by all replicas using the same key. The only way to enforce this limit without coordination is to restrict the number of distinct messages to $\frac{2^{32}}{n}$, where n is the maximum number of replicas that can use a single key. Thus, further limiting the number of encrypted messages. Assuming 1024 as an upper bound on the number of replicas, this leaves $\frac{2^{32}}{1024} = 2^{22} \approx 4.2 \times 10^6$ messages to each replica. Or, in other words, 7 weeks of coordination-free operation using one outbound message per second for each replica. Moreover, enforcing a limit on the number of replicas also requires coordination.

Fortunately, random nonces become practical with the very large nonce sizes supported by XChaCha20-Poly1305 [3]. The use of 192-bit nonces allows $2^{80} (\approx 10^{24})$ messages to be encrypted with a nonce collision probability of 2^{-32} [3]. Putting this in context: If every possible of the 2^{32} IPv4 devices is encrypting messages at the rate of one message per *millisecond*, this leaves us with over 8900 years before we must rotate keys. Therefore, with random nonces, we only recommend XChaCha20-Poly1305.

5.2.3 Nonce Misuse-resistant AEAD Schemes

Nonce misuse-resistant authenticated encryption schemes, such as AES-GCM-SIV [19], aim to be secure even when a nonce is reused for the same key with a different message. Thus, in theory, they are good candidates for use with shared, long-lived keys. But these schemes also do have bounds on the number of messages that can be safely sent [22]. Moreover, they are not yet standardized and fully scrutinized, and, as discussed in Figure 12, an implementation of AES-GCM-SIV is not widely available; thus, we can not give clear recommendations.

6 Evaluation

In our interim qualitative assessments, our focus was on design considerations in Section 3 (e.g., reusability, flexibility, correctness) and security guarantees in Section 4 and Section 5. The question remains: What is the cost of the properties of our approach featured in the qualitative assessments. To assess this cost, we empirically evaluate our approach along the following research questions:

- RQ1: Is the performance of ARDTs – including encryption – good enough for use in local-first applications?
- RQ2: Are the space requirements of ARDTs using intermediaries acceptable?

We use two methods to explore each of these questions: A concrete case study that makes specific choices about the used ARDTs and a set of microbenchmarks that explore encARDTs more generally to uncover their behavior in multiple dimensions, in particular the overhead caused by encryption and intermediaries. The implementation is part of the REScala project, and all implementations and benchmarks – in addition to further case studies that explore different scenarios – can be found on the project website¹². The case study evaluated here runs on the JVM (using a JavaFX UI) due to the generally better availability of tooling for

¹²<https://www.rescala-lang.com>

empirical evaluation, but our approach works for both the JVM and on the Web platform (integrating with various Scala-based Web UI frameworks). Unless otherwise noted, we use the following hardware and software setup for this evaluation.

CPU 2015 Intel Core i7-6700HQ (laptop CPUs are the most common for local-first software).

OS Arch Linux (Linux 5.16.16).

JRE We use the Java runtime OpenJDK 17.0.3.

Microbenchmarks For performance microbenchmarks, we use JMH¹³ the standard Java benchmarking tool. The time measurements we conduct have very stable runtime behavior, with a maximum relative error of 3%, thus we do not show error bars.

Libraries AEAD implementations are provided by Tink 1.6.1¹⁴, which uses hardware acceleration for AES variants, but not for XChaCha20-Poly1305. To serialize states, we use jsoniter-scala¹⁵ (the arguably fastest JSON serializer available on the JVM¹⁶).

6.1 Case Study

We implement the popular to-do list example as a JavaFX GUI application. The application manages a list of to-dos, and the user may add entries containing arbitrary text, mark to-dos as completed, change their text, or delete to-dos completely. Its correctness and consistency properties are: added to-dos remain until deleted, and all users see the same to-dos in the same order. The interactions and properties of the case study touch on most of the complexity in the design space of local-first applications. Furthermore, the state of the to-do list – a potentially ordered set of changeable entries – is complex enough to demonstrate the need for composed data types.

We experienced no limitations in implementing the to-do list application with ARDTs and encARDTs. The prototype makes heavy use of the composability of ARDTs. Concretely, the to-do list uses an add-wins last-writer-wins map for its primary state. This is a composition out of a tombstone-free add-wins set [8] and a last-writer-wins register. When two users edit the same to-do entry, a deterministic decision keeps one of the edits and the other is discarded. Changes to the primary state are normally triggered by the UI library (e.g., a button click handler), but the UI is replaced by our benchmark infrastructure. The handlers for each change are similar to the example in Figure 5. Each handler uses a corresponding operator on the to-do entries (the add-wins-last-writer-wins map) to compute the delta of the new application state. The delta is passed to the `send` operator of the dotted encARDT, and the operator computes its own delta that is in turn passed to the message dissemination implementation (a custom one for benchmarking the transferred data). In addition, there is a notification API (not discussed in the paper) in the message dissemination module that executes a handler whenever a change happens (caused locally or remotely), which triggers the UI to update and show the new state.

To answer our research questions, we run a deterministic simulation of the to-do list. Our simulation uses a single intermediary and simulates a total of one million operations that add, modify, and remove to-do entries (see Subsection 6.2 for a discussion of concurrent operations and multiple intermediaries). A million operations correspond to about 11 days of usage, with an interaction per second. We include serialization, encryption, and other application logic in the simulation. We omit physical network, storing the state on disk, or rendering the

¹³<https://openjdk.java.net/projects/code-tools/jmh/>

¹⁴<https://developers.google.com/tink>

¹⁵<https://github.com/plokhotnyuk/jsoniter-scala>

¹⁶<https://plokhotnyuk.github.io/jsoniter-scala/>

graphical UI, as their performance is not part of our contributions. The simulation follows a randomly generated trace of operations: adding to-dos, marking to-dos as completed, and deleting batches of the 30 oldest to-dos. To-dos are added and completed individually, but deleted in batches to reflect the expected usage of the application, which has a “remove all completed to-dos” button, but no methods of batch insertion or completion.

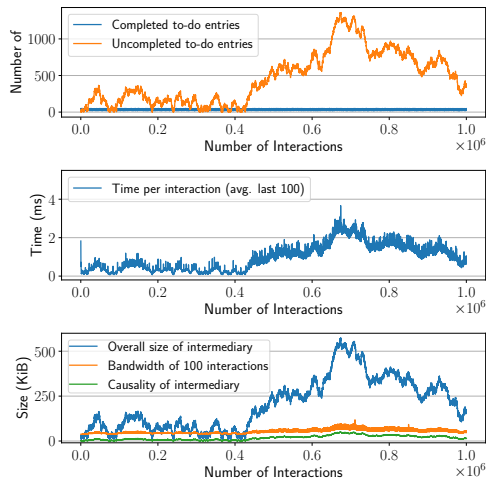
The top plot of Figure 13 shows the runtime behavior of the simulation. The x-axis represents abstract time as the number of executed interactions and the graphs show the respective state of the application, i.e., the number of open and completed to-do entries.

RQ1. Time overhead is presented in Figure 13 (middle plot). It shows the runtime per interaction (measured in batches of 100 interactions). This time includes executing the operator locally, merging it into the local state, serializing then encrypting and sending the delta, merging the encrypted delta into the encARDTs thus computing subsumption, and replicating the encARDTs to the intermediary. The spike in the beginning is due to the warm-up of the JVM. Otherwise, the overall runtime is proportional to the size of the current application state, because tasks – such as merging the add-wins-map, computing subsumption for existing deltas, and the application logic – linearly depend on the number of to-do entries. We believe that staying within 3 ms per operation is reasonable. While further optimizations are certainly possible, there is no indication that our core architecture has prohibitive costs for local-first applications.

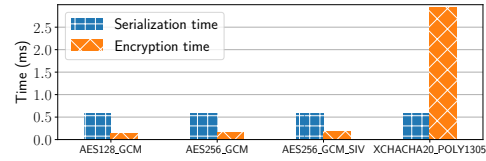
RQ2. Space overhead is presented in Figure 13 (bottom). Note that we show the accumulated bandwidth of 100 interactions (i.e., 100 deltas), because the size would otherwise not be visible at the scale of the figure. In summary, we observe that the total data stored at the intermediary has a linear relation to the actual size of the application state and grows and shrinks accordingly. We want to point out that neither the state, nor the causality metadata increases over time. While encARDTs require that we store information about subsumed deltas indefinitely (the set of subsumed dots), it is stored as efficient ranges that only grow with the number of replicas, concurrent operations, and current size of the data set, but not with the number of total interactions over time. The data transferred (used bandwidth) between the replica and the intermediary remains mostly constant because transfer time depends on the size of deltas, which are largely unaffected by the size of the application state. The slight increase in bandwidth is because each removal delta includes causality information in the encARDT that grows with the amount of currently non-removed entries. We show the difference to using a trusted intermediary (i.e., no encARDT) in Appendix A.6, which requires less space due to the impact of encrypted deltas discussed in Subsection 6.2. In conclusion, we consider the size demand and required bandwidth of ARDTs adequate for the local-first scenario.

6.2 Microbenchmarks

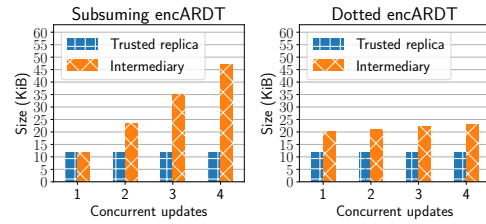
We perform microbenchmarks to acquire data points that the case study does not exhibit. Specifically, we investigate the isolated overhead of encryption as well as the effect of concurrent operations (e.g., due to multiple intermediaries) on the required storage size. We use the same add-wins last-writer-wins map (AWLWWMap) that was used for the to-do list case study. Yet, the results are independent from the concrete choice of ARDT, because for the microbenchmarks only the serialized size of the state matters, which we give explicitly.



■ **Figure 13** To-do list case study measurement results.



■ **Figure 14** Encryption vs. serialization time for AWLWMap states of 256 KB.



■ **Figure 15** State size when storing concurrent encrypted messages.

RQ1: Time overhead of encARDTs. We measure how long it takes to prepare the serialized bytes compared to the time for encrypting those bytes via an encARDT. The results in Figure 14 show the difference (encryption time vs. serialization time) for different AEAD schemes, and for a payload of 256 KiB (1,000 to-do entries). Hardware accelerated AES has an overhead of a fraction of a millisecond. XChaCha20-Poly1305 is designed to be efficiently implemented in software [3], thus should be considered for systems where no hardware accelerated encryption is available. Even if it does not benefit from hardware acceleration it has an overhead of less than 3 ms.

To put these numbers into context, consider the relative sizes of operations. The full state of a to-do list with 1000 to-do entries serializes into a 256 KiB state. The dotted encARDT requires an additional 0.16 ms for encryption. The serialization of the state alone takes 0.67 ms. Sending data over the network has expected latencies of 0.1~100 ms. Receiving and processing the data on the other replica and displaying the result adds a minimum of 7~33 ms due to typical refresh rates of monitors. In summary, we consider the time overhead of encARDTs to be negligible compared to all other parts of the synchronization process.

RQ2: Space overhead of encARDTs. Intermediaries cannot merge states that are created concurrently by different replicas, due to the limitations of the plaintext metadata. The overhead depends on the encARDT and the number concurrent operations. Figure 15 shows the space requirement of storing 1 to 4 concurrent updates using a subsuming encARDT (left) and a dotted encARDT (right). The base size of the stored ARDT is an AWLWMap with 96 (+1 to +4 added) entries requiring about 14 KiB. Any trusted replica (in blue) can always merge any received updates, thus the total stored size does not grow noticeably.

For the intermediary, however, we observe that the size of subsuming encARDT (left subfigure) grows linearly with each concurrent update. We expected this result as each update contains the full state to be stored, and the timestamps of the four updates are incomparable, because each full state differs in exactly one item, thus they do not subsume each other. For the dotted encARDT (right subfigure), the stored state is larger than the state of the trusted replica, because each delta is stored separately, which introduces a constant overhead per

delta. However, each concurrent update only marginally increases the state size to store the single additional delta. Note that the number of concurrent operations is typically limited by the number intermediaries, because once a replica is connected to an intermediary the next operations of the replica will merge and subsume concurrent operations.

In general, storing only deltas at intermediaries has a fixed overhead, but avoids large storage increases for concurrent updates. Which strategy is more suitable depends on how reliable connections are, and how many intermediaries are part of the system, because both unreliability and more intermediaries introduce more concurrency. In summary, we believe that a wide range of potential use cases are covered by the presented encARDTs. If other behavior is required, new variants of encARDTs with different subsumption strategies can be used.

7 Related Work

Programming methods for local-first software. Two popular general-purpose CRDT implementations that can be integrated into applications are `automerger`¹⁷ (loosely based on a paper by Kleppmann et al. [24]) and `Yjs`¹⁸ [37]. Both libraries are based on the operation-based variant of CRDTs. They run in the same process as the application and provide the latter with an API to update and query a single JSON document (a nested tree structure). The intended way to use the API is to have developers convert their application state into the JSON structure, with no further customization of available operations.

`REScala` [34] provides programming support for local-first applications. It integrates off-the-shelf CRDTs with functional reactive programming, the latter being a very common approach to UI and thus local-first applications. The rationale for the integration is that reactive applications are practically not limited by the monotonicity restriction of CRDTs, because user interactions are monotonic by nature: Users can only press, click, and touch keys and buttons and not “unpress” a prior action. However, `REScala` assumes that the developer provides CRDTs with suitable operations, and does not consider encryption [33, 35].

Similar to our work, other systems for local-first software consider message dissemination as an orthogonal concern that depends on the concrete network environment. `Yjs`, and `automerger`, provide default implementations to be ready to use, while `REScala` defers to `ScalaLoc` [58] – a library that abstracts over communication implementations. None of the approaches considers network security beyond encrypted direct connections. Almeida et al. and Enes et al. [2, 15] show how to achieve causal consistency for delta CRDTs independently of the underlying network. This is done by providing an additional layer on top of another message dissemination algorithm. While they do not consider security, their approach is similar in that they separate different concerns in the message dissemination.

Security in replicated systems. Prego et al. [39] survey CRDTs in the geo-distributed setting and point out the need for future research on security. They observe that replicas are vulnerable to harmful operations of other replicas, and that authentication and encryption between replicas is insufficient in the geo-distributed setting. This is because the trusted entities in that setting are the clients (end users) that issue operations to a replica (in the cloud). They conclude that end-to-end security between clients is (nearly) impossible in the existing architecture, and argue for “moving computations to the edge”. Moving computation to the edge, while replicating state in the cloud, is exactly what ARDTs enable.

¹⁷<https://github.com/automerger/automerger>

¹⁸<https://github.com/yjs/yjs>

Barbosa et al. [4] implement an approach that keeps the client/replica split while proving some guarantees to clients. The clients use customized solutions from the space of homomorphic encryption to secure their data before storing it in a distributed database (AntidoteDB [50]), which then handles replication. This approach has the major shortcoming that all the cryptographic constructions are specific to individual CRDTs. Moreover, they only target *honest-but-curious* adversaries, which is an assumption where the attacker is bound to service-level agreements, and only interested in secretly extracting information (i.e., a cloud service provider hosting the database). Crucially, this entails that an adversarial provider could modify data, because operations cannot be authenticated.

High-level cryptographic APIs. We do believe that encARDTs offer an advantage even when used with simple direct connections. Security is often treated as an afterthought, and it has been shown that leaving this task of using crypto solutions to application developers often leads to insecure systems [16, 38]. The correct usage of cryptographic components is challenging in general [36], with 84 % of Apache projects containing cryptographic misuses [41]. Especially developers of end-user applications seem to have a hard time, with more than 95 % of android applications that use a cryptographic API using it incorrectly [27]. High-level abstractions with built-in cryptographic features are considered as an effective solution to support developers with writing secure software [1, 18, 32]. With encARDTs, we bring high-level cryptographic APIs to local-first applications, thus reducing the potential for misuse. In addition, encARDTs define encryption of data structures, not encryption of connections, which is better suited to the flexible dynamic connections of local-first applications.

Identities and attackers. Security and authentication of our approach require a shared secret between trusted parties. If secrets are shared with untrusted parties, our approach does not provide additional guarantees. Sanjuan et al. [47] and Kleppmann [23] investigate settings where other replicas are not trusted. They argue that CRDTs are well suited to detect Byzantine faults at the eventual consistency layer. Specifically, by including cryptographic hashes of the causal history of each change, it is possible to discard and detect messages from misbehaving replicas. We believe that these solutions also apply to ARDTs due to their similarity to CRDTs. However, even when using these solutions an attacker may still execute consistent but undesirable actions, and is able to read the system state.

To prevent undesirable actions, we need a way to manage and enforce access policies over time. EncARDTs use a shared secret to define the current set of trusted replicas, and it is possible to rotate this key to change the set of trusted replicas. Rault et al. [42] propose how to manage access control itself as a CRDT, thus answering the question of who should have access to the shared secret. Truong et al. [55] discuss authentication of the log of operations in an RDT, which allows replicas to identify and attribute tampering carried out by replicas with full access. Kollmann et al. [26] propose a solution to compress such authentication information within a snapshot of an RDT. This also allows them to keep the exact history of changes hidden from newly joined replicas by leveraging coordination-free authentication of snapshots. We believe that these approaches could be adapted for the use with ARDTs.

8 Conclusions and Future Work

Local-first applications address several weaknesses of a centralized software architecture. But designing applications with consistent replication is challenging for application developers, because it requires expertise in several system-level concerns such as consistency, networking,

and security protocols. Existing solutions such as CRDTs provide consistency “out-of-the-box”, but have several shortcomings otherwise. First, they do not integrate well into the application design process: Application developers have to map application-specific data models to CRDTs, which are only available “off-the-shelf” in the form of databases [50] or libraries with a fixed API [24, 37]. Designing application state based on a fixed set of operations is known to cause design issues [11]. Second, off-the-shelf systems do not support heterogeneous network environments, and authenticity and confidentiality is considered as an afterthought at best.

The foundation of our solution to the above gaps is an approach for systematic, modular, and extensible design of algebraic replicated data types (ARDTs). The approach provides the same guarantees as CRDTs, but as a modular and extensible library that embraces algebraic data types, which are widely used to model application state. This approach facilitates the integration of ARDTs into existing programming models and existing network runtimes. Further, our solution provides confidentiality and authenticity by design. Specifically, we presented a family of encrypting ARDTs for different network requirements. Each such encARDT wraps around the data of an ARDT and secures the data independently of how messages are disseminated, with specific support to transmit data over untrusted intermediaries. Using our encARDTs, the application data is authenticated and encrypted, while retaining coordination freedom and preventing common misuses of cryptographic primitives. A significant partial result of the above is, that while current AEAD schemes theoretically require coordination due to the uniqueness constraint on the nonces, it is possible to avoid coordination for long enough to make them applicable in a coordination-free setting. Specifically, this result applies to all other approaches for local-first software that could adapt our techniques to encrypt and authenticate their network communication.

Our evaluation shows that we can implement typical local-first applications efficiently and that any ARDT can be securely disseminated. The performance overhead is only a small fraction of the existing dissemination cost. The additional storage requirement is limited by the amount of concurrent changes in the worst case and can be minimized further by including more precise metadata. Moreover, the storage requirement does not increase indefinitely, as ARDTs makes it possible to remove data that is no longer needed by the application logic. Together, the results of the experiments show that it is feasible to use the proposed solution in practice.

A remaining issue – common to all encrypted synchronization techniques – is that it needs to leak metadata to enable efficient dissemination of messages. However, because our approach is resilient to poor network conditions including reordering, delay, and duplication of messages, we believe that many common mitigation techniques can be applied without impeding normal operations. Such mitigations include sending fake data to make metadata less usable, or routing data on multiple intermediaries such that no single one has a full view of the system. We may also be able to apply concepts from homomorphic encryption or secure enclaves to enable intermediaries to learn which states subsume each other, without gaining any further insight into the exact metadata of each message.

Finally, it is noteworthy that besides solving the practical problem of ensuring the integrity and authenticity of replicated state in the presence of untrusted replicas, encARDTs also represent the novel concept of RDT-based implementations of what would classically be seen as a (network) protocol. An encARDT addresses protocol concerns such as transparency of encrypting and decrypting transferred data, which messages are important (and must be retransmitted), and which ones have been superseded by newer messages. Crucially, these concerns are separated from concrete issues concerning physical networks such as message

losses, retries and retransmission delays, or splitting large packages. These concerns are specific to each communication platform and are handled by concrete message dissemination modules. As a future expansion on this concept it could be possible to implement other concerns of network protocols as ARDTs. A concrete example is message delivery in causal order, which can be achieved by attaching ordering information to each message [2]. Such an ARDT would wrap another ARDT, similar to how an encARDT works, but use its operator to present a state merged in causal order (temporarily ignoring messages that were received out of order).

References

- 1 Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. Comparing the usability of cryptographic apis. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 154–171, 2017. doi:10.1109/SP.2017.52.
- 2 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018. doi:10.1016/j.jpdc.2017.08.003.
- 3 Scott Arciszewski. XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. Internet-Draft draft-irtf-cfrg-xchacha-03, Internet Engineering Task Force, 2020. Work in Progress. URL: <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-xchacha-03>.
- 4 Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. Secure conflict-free replicated data types. In *International Conference on Distributed Computing and Networking 2021, ICDCN '21*, pages 6–15, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3427796.3427831.
- 5 Lars Baumgärtner, Jonas Höchst, and Tobias Meuser. B-dtn7: Browser-based disruption-tolerant networking via bundle protocol 7. In *2019 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, pages 1–8, 2019. doi:10.1109/ICT-DM47966.2019.9032944.
- 6 Lars Baumgärtner, Patrick Lieser, Julian Zobel, Bastian Bloessl, Ralf Steinmetz, and Mira Mezini. Loragent: A dtn-based location-aware communication system using lora. In *2020 IEEE Global Humanitarian Technology Conference (GHTC)*, pages 1–8, 2020. doi:10.1109/GHTC46280.2020.9342886.
- 7 Daniel J. Bernstein. Extending the salsa20 nonce. In *Workshop Record of Symmetric Key Encryption Workshop 2011*, 2011. URL: <https://cr.yp.to/snuffle/xsalsa-20110204.pdf>.
- 8 Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. Research Report RR-8083, INRIA, October 2012. URL: <https://inria.hal.science/hal-00738680>.
- 9 Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, 2016. URL: <https://www.usenix.org/conference/woot16/workshop-program/presentation/bock>.
- 10 Russell Brown. Vector clocks revisited, 2015. Online; accessed 18 October 2021. URL: <https://riak.com/posts/technical/vector-clocks-revisited/index.html>.
- 11 Tse-Hsun Chen, Weiyi Shang, Jinqiu Yang, Ahmed E. Hassan, Michael W. Godfrey, Mohamed N. Nasser, and Parminder Flora. An empirical study on the practice of maintaining object-relational mapping code in java systems. In Miryung Kim, Romain Robbes, and Christian Bird, editors, *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 165–176. ACM, 2016. doi:10.1145/2901739.2901758.
- 12 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 1. ACM, 2012. doi:10.1145/2391229.2391230.

- 13 Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association. URL: <https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router>.
- 14 Morris J. Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, National Institute of Standards and Technology, 2007. doi:10.6028/nist.sp.800-38d.
- 15 Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. Efficient synchronization of state-based crdts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 148–159. IEEE, 2019. doi:10.1109/ICDE.2019.00022.
- 16 Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, pages 50–61, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2382196.2382205.
- 17 Google. Authenticated encryption with associated data (aead). Online; accessed 12 October 2021. URL: <https://developers.google.com/tink/aead>.
- 18 Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016. doi:10.1109/MSP.2016.111.
- 19 Shay Gueron and Yehuda Lindell. GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle per Byte. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, pages 109–119, New York, NY, USA, October 2015. Association for Computing Machinery. doi:10.1145/2810103.2813613.
- 20 Christoph Hagen, Christian Weinert, Christoph Sendner, Alexandra Dmitrienko, and Thomas Schneider. All the numbers are US: large-scale abuse of contact discovery in mobile messengers. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. doi:10.14722/ndss.2021.23159.
- 21 Joseph M. Hellerstein and Peter Alvaro. Keeping calm: When distributed consistency is easy. *Commun. ACM*, 63(9):72–81, 2020. doi:10.1145/3369736.
- 22 Antoine Joux. Nonce misuse-resistant authenticated encryption, 2019. doi:10.17487/RFC8452.
- 23 Martin Kleppmann. Making CRDTs Byzantine fault tolerant. In *9th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2022*, pages 8–15. ACM, April 2022. doi:10.1145/3517209.3524042.
- 24 Martin Kleppmann and Alastair R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017. doi:10.1109/tpds.2017.2697382.
- 25 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019*, pages 154–178, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.
- 26 Stephan A Kollmann, Martin Kleppmann, and Alastair R Beresford. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2019(3):210–232, July 2019. doi:10.2478/popets-2019-0044.
- 27 Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An extensible approach to validating the correct usage of cryptographic APIs. *IEEE Transactions on Software Engineering*, 47(11):2382–2400, 2019. doi:10.1109/TSE.2019.2948910.
- 28 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.
- 29 Paul J. Leach, Rich Salz, and Michael H. Mealling. A universally unique identifier (uuid) urn namespace. RFC 4122, 2005. doi:10.17487/RFC4122.

- 30 Libsodium Project. Aes256-gcm. Online; accessed 14 October 2021. URL: https://libsodium.gitbook.io/doc/secret-key_cryptography/aead/aes-256-gcm.
- 31 David McGrew. An interface and algorithms for authenticated encryption. RFC 5116, 2008. doi:10.17487/RFC5116.
- 32 Kai Minderhann, Philipp Keck, and Stefan Wagner. How usable are rust cryptography apis? In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 143–154, 2018. doi:10.1109/QRS.2018.00028.
- 33 Ragnar Mogk. *A Programming Paradigm for Reliable Applications in a Decentralized Setting*. PhD thesis, Technische Universität Darmstadt, Darmstadt, March 2021. doi:10.26083/tuprints-00019403.
- 34 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant distributed reactive programming. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPICs*, pages 1:1–1:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ECOOP.2018.1.
- 35 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019. doi:10.1145/3360570.
- 36 Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: why do java developers struggle with cryptography APIs? In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 935–946. ACM, 2016. doi:10.1145/2884781.2884790.
- 37 Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In Philipp Cimiano, Flavius Frasinca, Geert-Jan Houben, and Daniel Schwabe, editors, *Engineering the Web in the Big Data Era*, pages 675–678. Springer International Publishing, 2015. doi:10.1007/978-3-319-19890-3_55.
- 38 Marten Oltrogge, Nicolas Huaman, Sabrina Amft, Yasemin Acar, Michael Backes, and Sascha Fahl. Why eve and mallory still love android: Revisiting TLS (In)Security in android applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4347–4364. USENIX Association, August 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/oltrogge>.
- 39 Nuno Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types crdts. In Sherif Sakr and Albert Zomaya, editors, *Encyclopedia of Big Data Technologies*, pages 1–10. Springer International Publishing, 2018. doi:10.1007/978-3-319-63962-8_185-1.
- 40 Nuno Preguiça, Carlos Bauqero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 335–336, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2332432.2332497.
- 41 Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3345659.
- 42 Pierre-Antoine Rault, Claudia-Lavinia Ignat, and Olivier Perrin. Distributed access control for collaborative applications using CRDTs. In *9th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2022, pages 33–38. ACM, April 2022. doi:10.1145/3517209.3524826.
- 43 Eric Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, 2018. doi:10.17487/RFC8446.

- 44 Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security, CCS '02*, pages 98–107, New York, NY, USA, November 2002. Association for Computing Machinery. doi:10.1145/586110.586125.
- 45 Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2004. doi:10.1007/978-3-540-25937-4_22.
- 46 Joseph A. Salowey, David McGrew, and Abhijit Choudhury. Aes galois counter mode (gcm) cipher suites for tls. RFC 5288, 2008. doi:10.17487/RFC5288.
- 47 Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. Merkle-CRDTs: Merkle-DAGs meet CRDTs. *CoRR*, April 2020. arXiv:2004.00107.
- 48 Sebastian Schildt, Tim Lüdtkke, Klaus Reinprecht, and Lars Wolf. User study on the feasibility of incentive systems for smartphone-based dtms in smart cities. In *Proceedings of the 2014 ACM International Workshop on Wireless and Mobile Technologies for Smart Cities, WiMobCity '14*, pages 67–76, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2633661.2633662.
- 49 Bruce Schneier. *Applied cryptography – Protocols, algorithms, and source code in C, 2nd Edition*. Wiley, 1996. URL: <https://www.worldcat.org/oclc/32311687>.
- 50 Marc Shapiro, Annette Bieniusa, Nuno M. Pregoça, Valter Balesgas, and Christopher Meiklejohn. Just-right consistency: Reconciling availability and safety. *CoRR*, abs/1801.06340, 2018. doi:arXiv.1801.06340.
- 51 Marc Shapiro, Nuno Pregoça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, 2011. URL: <https://hal.inria.fr/inria-00555588>.
- 52 Marc Shapiro, Nuno Pregoça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-24550-3_29.
- 53 Milan Stute, Florian Kohnhauser, Lars Baumgartner, Lars Almon, Matthias Hollick, Stefan Katzenbeisser, and Bernd Freisleben. RESCUE: A resilient and secure device-to-device communication framework for emergencies. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2020. doi:10.1109/TDSC.2020.3036224.
- 54 Chengzheng Sun. Reflections on collaborative editing research: From academic curiosity to real-world application. In Weiming Shen, Pedro Antunes, Nguyen Hoang Thuan, Jean-Paul A. Barthès, Junzhou Luo, and Jianming Yong, editors, *21st IEEE International Conference on Computer Supported Cooperative Work in Design, CSCWD 2017, Wellington, New Zealand, April 26-28, 2017*, pages 10–17. IEEE, 2017. doi:10.1109/CSCWD.2017.8066663.
- 55 Hien Thi Thu Truong, Claudia-Lavinia Ignat, and Pascal Molli. Authenticating operation-based history in collaborative systems. In *17th ACM International Conference on Supporting Group Work, GROUP 2012*, pages 131–140. ACM, October 2012. doi:10.1145/2389176.2389197.
- 56 Jelle van den Hooff, David Lazar, Matei Zaharia, and Nikolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15*, pages 137–152, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2815400.2815417.
- 57 Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pages 1313–1328, New York, NY, USA, October 2017. Association for Computing Machinery. doi:10.1145/3133956.3134027.
- 58 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with scalaloci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, 2018. doi:10.1145/3276499.

```

70 inline def derived[S<:Product](using pm:ProductOf[S]):Lattice[S]=
71   val lattices =
72     summonAll[Tuple.Map[pm.MirroredElemTypes, Lattice]]
73     .toArray.map(_.asInstanceOf[Lattice[Any]])
74   new Lattice[S]:
75     def merge(left: S, right: S): S = pm.fromProduct(
76       new Product {
77         def productElement(i: Int): Any =
78           lattices(i).merge(left.productElement(i),
79                             right.productElement(i))
80       })

```

■ **Figure 16** Automatic derivation of lattice instances for product types.

A Appendix

A.1 Map Merge is Correct

Proof. Given that K is the set of all keys (replica ids), x_k is a lookup of key k in map x that returns 0 if the key is not present, $\{k \rightarrow v\}_{k \in K}$ constructs a new map that associates the key k to the value v , that m_0 is a correct merge function for the values stored in the map, and $m(x, y) = \{k \rightarrow m_0(x_k, y_k)\}_{k \in K}$ is the implementation of the merge function. All three proofs are calculations that first expand the definition of m , then use the respective property of the m_0 function, and finally use the reverse definition of m (except in the idempotence case which is already done).

$$\begin{aligned}
 \text{Commutative: } m(x, y) &= \{k \rightarrow m_0(x_k, y_k)\}_{k \in K} \\
 &= \{k \rightarrow m_0(y_k, x_k)\}_{k \in K} = m(y, x)
 \end{aligned} \tag{1}$$

$$\begin{aligned}
 \text{Associative: } m(m(x, y), z) &= \{k \rightarrow m_0(m_0(x_k, y_k), z_k)\}_{k \in K} \\
 &= \{k \rightarrow m_0(x_k, m_0(y_k, z_k))\}_{k \in K} = m(x, m(y, z))
 \end{aligned} \tag{2}$$

$$\begin{aligned}
 \text{Idempotent: } m(x, x) &= \{k \rightarrow m_0(x_k, x_k)\}_{k \in K} \\
 &= \{k \rightarrow x_k\}_{k \in K} = x
 \end{aligned} \tag{3}$$

◀

A.2 Derived Product Merge Implementation

We elaborate on the technical details of the implementation of method `derived` in Figure 16. The method is marked `inline` to make use of compile-time meta programming, which we use to acquire lattice instances of the individual components of the product, specifically, the `summonAll` method used later. The `using` keyword asks the compiler to provide a product mirror (named `pm`) for the product type `S` to allow inspection of the components of `S`.

The `summonAll` method in Line 72, similar to the `using` keyword, “summons” instances provided by the given keyword based on their types. Specifically, the type we request are the component types (`pm.MirroredElemTypes`) of our product mapped to the `Lattice` type. As an example, the component types of our `MyData` class returns the type `(A, B)` and mapping

that onto the lattice type results in the type (`Lattice[A]`, `Lattice[B]`) which is the type for which we “summon” the instances. The result is a tuple of typed lattice instances, but we throw away all type information (Line 73) and rely on the fact that all used products have the same component type at the same structural position.

Having computed lattice instances of the components, we create a new instance of the lattice trait (Line 74). The merge function of that instance (defined in Line 75) uses the `pm.fromProduct` helper to generically create a new instance of the result product `S` (e.g., a new instance of our `MyData` class) in Line 76. The parameter to `pm.fromProduct` essentially assigns each component at index `i` (`productElement` in Line 76) the result of using merge function `i` to merge the left and right components at position `i`.

The technical challenges of generating merge functions for arbitrary products are mostly related to practical concerns in the programming language.

A.3 Derived Product Merge is Correct

Proof. Given that K is the set of product indices (this would be the field names of a case class), x_k is a lookup of index k in product x , the syntax $\{k \rightarrow v\}_{k \in K}$ constructs a new product of correct type that associates the index k to the value v , each component type at index k has a merge function m_k , and $m(x, y) = \{k \rightarrow m_k(x_k, y_k)\}_{k \in K}$ is the implementation of the merge function for the product. We show that m is commutative, associative, and idempotent. All three proofs are calculations that first expand the definition of m , then use the respective property of the component merge functions, and finally use the reverse definition of m (except in the idempotence case which is already done).

$$\begin{aligned} \text{Commutative: } m(x, y) &= \{k \rightarrow m_k(x_k, y_k)\}_{k \in K} \\ &= \{k \rightarrow m_k(y_k, x_k)\}_{k \in K} = m(y, x) \end{aligned} \tag{4}$$

$$\begin{aligned} \text{Associative: } m(m(x, y), z) &= \{k \rightarrow m_k(m_k(x_k, y_k), z_k)\}_{k \in K} \\ &= \{k \rightarrow m_k(x_k, m_k(y_k, z_k))\}_{k \in K} = m(x, m(y, z)) \end{aligned} \tag{5}$$

$$\begin{aligned} \text{Idempotent: } m(x, x) &= \{k \rightarrow m_k(x_k, x_k)\}_{k \in K} \\ &= \{k \rightarrow x_k\}_{k \in K} = x \end{aligned} \tag{6}$$

◀

A.4 Naive encARDT is Transparent

Proof. We show that for any subset of states $c \subset S$ sending (encrypting) and recombining (decrypting and merging) the set c is equivalent to merging the set c directly. This uses the secret key k , the merge function m_S for states in S , the merge function $m_e = \text{union}$ of the encARDT, the encrypt e_k and decrypt d_k function with $d_k(e_k(s)) = s$, the send function $\text{send}_k(s) = \{e_k(s)\}$, and the recombine function $\text{rec}_k(c) = m_S(\{d_k(s) \mid s \in c\})$. The proof is done by expanding the above definitions (highlighted in blue) when appropriate.

$$\begin{aligned}
& \text{rec}_k(m_e(\{\text{send}_k(s')|s' \in c\})) \\
&= m_S(\{d_k(s)|s \in m_e(\{\text{send}_k(s')|s' \in c\})\}) \quad \text{def of rec} \\
&= m_S(\{d_k(s)|s \in m_e(\{\{e_k(s')\}|s' \in c\})\}) \quad \text{def of send} \\
&= m_S(\{d_k(s)|s \in \{e_k(s')|s' \in c\}\}) \quad \text{def of } m_e \tag{7} \\
&= m_S(\{d_k(e_k(s))|s \in c\}) \quad \text{simplify set ops} \\
&= m_S(\{s|s \in c\}) \quad \text{def of } e_k \text{ and } d_k \\
&= m_S(c)
\end{aligned}$$

◀

A.5 Subsuming encARDT is Transparent

Proof. Given a secret key k , a subset of states $c \subset S$, individual states s, x, y, z , a merge function m_S for states in S , associated data for each state a_s where $a_x \leq a_y$ if $m_S(x, y) = y$, the filter function $f(c) = \{x \in c | \nexists y \in c : a_x < a_y\}$, the merge function $m_e(c) = f(\bigcup(c))$ of the encARDT, the encrypt e_k the decrypt d_k function with $d_k(e_k(s)) = s$, the current encrypted states c_e , the send function $\text{send}_k(c_e, s) = \{e_k(m_S(\text{rec}_k(c_e), s))\}$, and the recombine function $\text{rec}_k(c_e) = m_S(\{d_k(s)|s \in c_e\})$.

It holds that filtering distributes over union $f(p \cup q) = f(f(p) \cup q)$, because all elements of $f(p)$ are larger or equal to all elements in p , so filtering them out first does not change the result of $f(p \cup q)$.

Filter distributes over decryption, i.e., $\{d_k(s)|s \in f(c)\} = f(\{d_k(s)|s \in c\})$, because filtering is defined on associated data which is also available in the encrypted state.

It holds that filtering is subsumed by merging $m_S(f(c)) = m_S(c)$, because for each removed element $r \in p \setminus f(p)$ it is subsumed by one of the remaining elements $q \in f(p)$ thus merging it again makes no difference $m_S(r, p) = p$.

We first show that the merge function of the encARDT m_e is associative, idempotent, and commutative. Note, that up until now, we have proven a slightly stronger version of idempotence that requires less calculation, but we can not do so here, because the filtering function does not provide the stronger guarantee that $f(a) = a$, thus we only have $m(x, x) = f(x)$. Instead of strong idempotence, we prove that $m(m(x, y), y) = m(x, y)$, that is, merging y multiple times still makes no difference, but we must merge at least once.

$$\text{Commutative: } m_e(x, y) = f(x \cup y) = f(y \cup x) = m_e(y, x) \tag{8}$$

$$\begin{aligned}
\text{Associative: } m_e(m_e(x, y), z) &= f(f(x \cup y) \cup z) = f(x \cup y \cup z) \\
&= f(x \cup f(y \cup z)) = m_e(x, m_e(y, z))
\end{aligned} \tag{9}$$

$$\begin{aligned}
\text{Idempotent: } m_e(m_e(x, y), y) &= f(f(x \cup y) \cup y) = f(x \cup y \cup y) \\
&= f(x \cup y) = m_e(x, y)
\end{aligned} \tag{10}$$

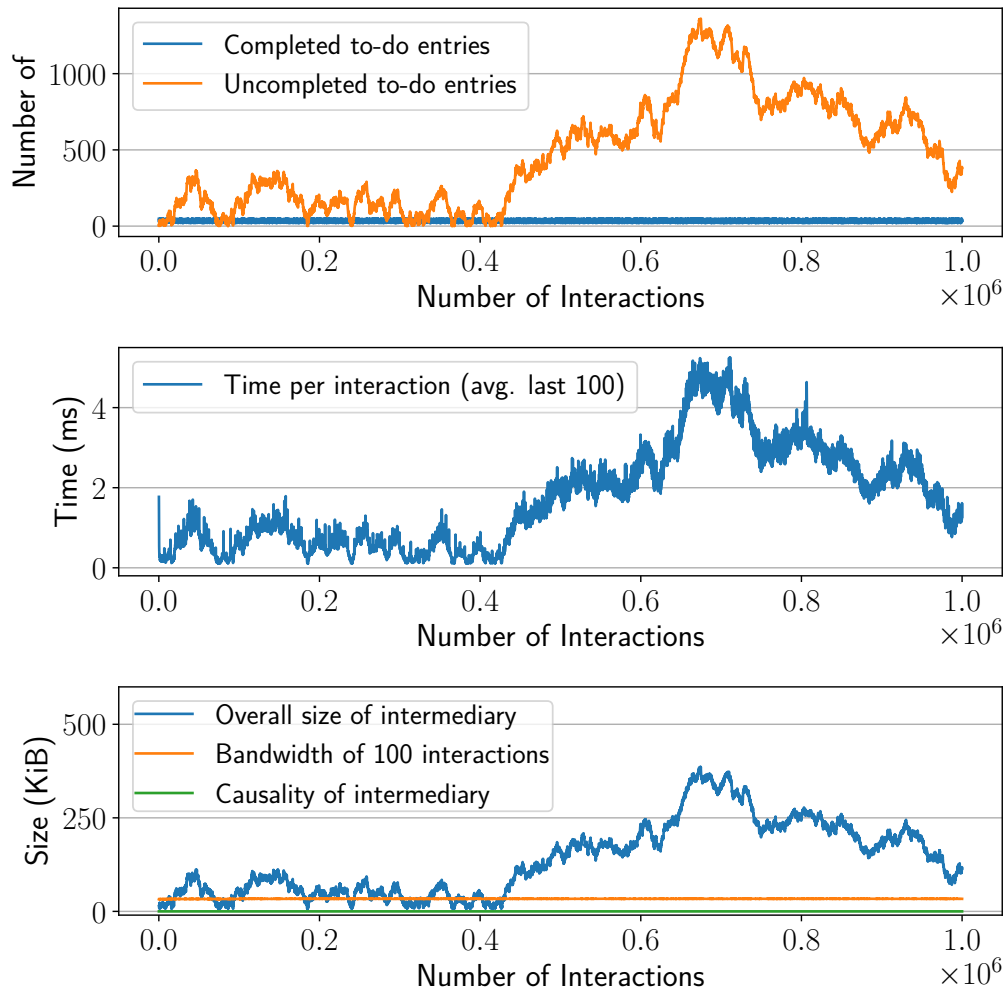
Finally, transparency of the subsuming encARDT, i.e., that receiving a set of send and filtered states is equivalent to merging those states directly. The applied definitions are listed and the changes highlighted in blue.

$$\begin{aligned}
& \text{rec}_k(m_e(\{\text{send}_k(c_e, s') | s' \in c\})) \\
&= m_S(\{d_k(s) | s \in m_e(\{\text{send}_k(c_e, s') | s' \in c\})\}) && \text{def of rec} \\
&= m_S(\{d_k(s) | s \in m_e(\{\{e_k(m_S(\text{rec}_k(c_e), s'))\} | s' \in c\})\}) && \text{def of send} \\
&= m_S(\{d_k(s) | s \in f(\{e_k(m_S(\text{rec}_k(c_e), s')) | s' \in c\})\}) && \text{def of } m_e \\
&= m_S(f(\{d_k(s) | s \in \{e_k(m_S(\text{rec}_k(c_e), s')) | s' \in c\}\})) && \text{filter distributes} \\
&= m_S(f(\{d_k(e_k(m_S(\text{rec}_k(c_e), s')) | s' \in c\})) && \text{simplify set ops} \\
&= m_S(f(\{m_S(\text{rec}_k(c_e), s') | s' \in c\})) && \text{def of } e_k \text{ and } d_k \\
&= m_S(\{m_S(\text{rec}_k(c_e), s') | s' \in c\}) && \text{filter subsumed} \\
&= m_S(\text{rec}_k(c_e), m_S(c)) && \text{merge properties} \\
&= m_S(m_S(\{d_k(e_k(s)) | e_k(s) \in c_e\}), m_S(c)) && \text{def of rec} \\
&= m_S(m_S(\{s | e_k(s) \in c_e\}), m_S(c)) && \text{decrypted} \\
&= m_S(\{s | e_k(s) \in c_e\} \cup c) && \text{merge properties}
\end{aligned} \tag{11}$$

◀

A.6 Case Study with Trusted Intermediary

Figure 17 shows the benchmark results for the to-do list case study when we trust the intermediaries and do not use an encARDT. The overall trends are similar, both the time per interaction and the size stored on the intermediary have a linear correlation to the current number of to-do entries. This is because those costs are inherent to the ARDT of the to-do list. There are notable differences. First, the overall runtime when using encARDTs is better (each interaction is faster), because merging the encARDT on the intermediary (i.e., pruning subsumed deltas) is faster than merging the to-do list on the intermediary (i.e., merging the two add-wins-last-writer-wins maps). The encryption overhead is negligible compared to that cost. Second, the overall size of the stored data on the trusted intermediary is smaller, because storing individual encrypted deltas requires more space as discussed in Subsection 6.2. Third, the client does not have to transmit any additional causality information and also does not create subsuming deltas that would reduce the overall size of an encARDT, but lead to larger deltas in some cases. This leads to a nearly constant bandwidth use, with small variations for the random difference between the relative amount of added, completed, and removed to-dos, as well as differences in to-do description lengths.



■ **Figure 17** To-do list case study measurement results with trusted intermediary.

Behavioural Types for Local-First Software


Roland Kuhn ✉ 

Actyx AG, Kassel, Germany

Hernán Melgratti ✉ 

University of Buenos Aires, Argentina

Conicet, Buenos Aires, Argentina

Emilio Tuosto ✉ 

Gran Sasso Science Institute, L'Aquila, Italy

Abstract

Peer-to-peer systems are the most resilient form of distributed computing, but the design of robust protocols for their coordination is difficult. This makes it hard to specify and reason about global behaviour of such systems.

This paper presents *swarm protocols* to specify such systems from a *global* viewpoint. Swarm protocols are projected to *machines*, that is *local* specifications of peers. We take inspiration from behavioural types with a key difference: peers communicate through an event notification mechanism rather than through point-to-point message passing. Our goal is to adhere to the principles of *local-first software* where network devices collaborate on a common task while retaining full autonomy: every participating device can locally make progress at all times, not encumbered by unavailability of other devices or network connections. This coordination-free approach leads to inconsistencies that may emerge during computations. Our main result shows that under suitable well-formedness conditions for swarm protocols consistency is eventually recovered and the locally observable behaviour of conforming machines will eventually match the global specification.

Our model elaborates on the Actyx industrial platform and provides the basis for tool support: we sketch an implemented prototype which proves this work a viable step towards reasoning about local-first and peer-to-peer software systems.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Distributed systems organizing principles; Software and its engineering → Distributed programming languages

Keywords and phrases Distributed coordination, local-first software, behavioural types, publish–subscribe, asynchronous communication

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.15

Related Version The extended version of this paper [27] contains more examples, a comparison between our model and *state machine replication* [32], more details on the Actyx middleware, and a discussion on the limitation of our approach.

Extended Version: <https://arxiv.org/abs/2305.04848>

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.14>

Funding Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, by MIUR project PRIN 2017FTXR7S *IT MATTERS* (Methods and Tools for Trustworthy Smart Systems), by the PRO3 MUR project Software Quality, and by the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA – Advanced Space Technologies and Research Alliance. *machine-runner* and *machine-check* partly funded by the European Union (TaRDIS, 101093006).

Acknowledgements The authors thank the anonymous reviewers for their useful and insightful comments and Daniela Marottoli for her help in the initial phase of this project.



© Roland Kuhn, Hernán Melgratti, and Emilio Tuosto;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 15; pp. 15:1–15:28



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

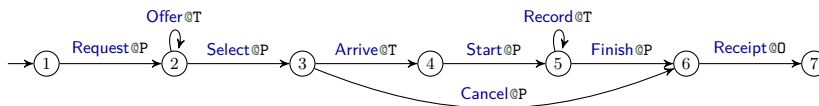


1 Introduction

Fully decentralised systems like peer-to-peer (P2P) networks are notoriously hard to design and analyse. A main challenge is to coordinate components so that the composed system exhibits the expected behaviour. As all decisions in such a system are made locally based on the available partial knowledge, the main problem is to specify which information is transferred to whom and when and how to interpret it, i.e. protocol design. The example we observed with Actyx is implemented in factory shop floor coordination, which is mission-critical: logistics robots compete for transport orders to move goods between machines and warehouses¹. For familiarity, we use taxi rides instead which are of the same shape.

► **Example 1.1** (Our running example). The fleet of a taxi company organises rides within a town using a P2P network. The main goal is to provide any registered passenger who requests a ride with some offers from taxis willing to perform the desired transportation; the passenger may pick one offer based on price and estimated time of arrival – followed by tracking the pickup, ride, and arrival – or cancel the ride. At the end of the trip the accounting office provides a receipt for the journey or cancellation. Note how there is some level of trust between the parties involved; we will treat this as a non-adversarial setting.

The structure of the expected interaction between the different peers (classified by *roles*) can be informally illustrated with the following diagram involving a passenger role P, a taxi role T, and an accounting office role as O.



When a passenger needs a ride (state 1 in the diagram above), they open an auction by executing a **Request**. Then, any taxi with capacity can proceed with an **Offer**. In this description, we do not make any assumption about the number of instances playing each role; in fact, we expect to have many taxis playing role T and hence many offers. The passenger ends the auction by using **Select** to pick a winner; note that we do not capture which taxi won the contract, instead we assume that only the winning taxi will perform actions after state 3 (we could model one selected branch per taxi and replicate states 3–6, but this only makes the example larger without additional insight).

The second phase starts with a race in state 3: the taxi begins the ride invoking **Arrive** while the passenger loses patience and uses the **Cancel** command to back out. The office will create a receipt in either case, but we must settle the dispute whether a ride happened. ─

The classic solution to such a problem would use a central database to present an up-to-date view of the respective data each participant is allowed to see. This solution avoids conflicts, maintains invariants, and steers the whole process by virtue of there being only one source of truth – one of the **Arrive** or **Cancel** commands would happen first and the

¹ This use-case has been implemented and is used in factories based on the Actyx Pond library, which uses the same event log replication mechanism as described in this article but does not model the interaction of differently shaped machines nor does it provide formal verification of the protocol. For a different factory the implementation and deployment is ongoing using the theory presented herein – developers report greater confidence and productivity than with established approaches. The main difficulty in factory logistics is reasoning about transient network partitions of mobile participants. Note that orchestrating the movement of goods on the shop floor is most critical for a factory’s success!

other would be rejected. It is well-known (cf. the CAP [16] and FLP [14] results) that such a solution suffers from unavailability if the system model includes network partitions, since either the database is localised and may be unreachable, or it is distributed and may need to reject requests to maintain consistency. With the advent of *conflict-free replicated data types (CRDTs)* [33] a different solution came into view: instead of avoiding conflicts through coordination, CRDTs provide a data model of such a structure that is conflict-free by construction. CRDTs facilitate that by demanding a join semi-lattice for the data structure, i.e. that there is a merge function that given two differently evolved states will compute a new state that represents the sum of all operations that were done to either input. Applied to Example 1.1 this would typically be achieved by systematically preferring either side of the choice at state 3, e.g. “cancellation always wins” (cf. the *add-wins set* CRDT). This illustrates that CRDTs are not well-suited for capturing and fairly resolving conflicts such as the one in our example. Nevertheless, the increasing research focus on coordination-free systems inspired the formulation of *local-first software* [25] in which all participants in a distributed system maintain full autonomy and control over their data. Besides the focus on agency and ownership, local-first principles can also be used to build software that is fully available and maximally resilient [26], where each participant can independently make decisions that are globally valid.

We propose a new way of approaching local-first software based on embracing conflict, recognising it, and finally reconciling it to reach eventual consensus [38]. Our model builds upon the middleware developed at Actyx [1], which provides a reliable durable pub-sub mechanism for event logs as well as a coordination-free total order of all events. A distributed system in our model is realised by a set of participants, dubbed *machines*, that can exhibit discordant behaviour and interact by broadcasting and reacting to *events*. Events are generated locally in response to the execution of *commands*, added to the *local log* and then propagated to other participants, to be merged into the log local at the recipient. We assume that events propagate asynchronously and that there is no traditional mechanism for coordination (like consensus or central nodes): machines liaise with each other purely on the basis of the events spreading in the system. We refer to such systems as *swarms*.

Each machine in a swarm implementing the scenario in Example 1.1 plays a role, i.e. it subscribes to a defined subset of event types and applies an assigned logic to interpret those. Depending on its local state (which it computes from its local log), any machine may decide to execute a command. For instance, a machine playing the role P may execute **Request**, which generates new events containing details of the request. Such events are appended to the local log of the machine and then propagated asynchronously to other machines that have subscribed to such events, e.g. the machines corresponding to taxis. After receiving such events, a taxi updates its local state and decides whether to place an offer for the ride. In such case, it executes **Offer**, which generates the events describing the offer, appends them to local log and propagates them to the rest of the system. The interaction described in Example 1.1 may proceed to completion in this way. Our swarms protocols specify the intended communications in an ideal run of the protocol, assuming that different sessions are independent from each other – we therefore do not represent sessions in our syntax.

Noteworthy, our computational model does not preclude the execution of conflicting commands. For instance, the race in state 3 of Example 1.1 allows two machines to generate their respective events and propagate them through the system. When receiving such events, each machine will be in charge of detecting and properly resolving the conflict. This is achieved by using the total order between events – interpreted as manifestation of (logical) time: the earliest event emitted after a choice decides which branch is taken (events from a

losing branch are ignored). Note that we do not assume knowledge of when the event log is complete, i.e. a machine cannot detect whether an event is globally the earliest after a choice; thus, computed local states may transiently diverge. As soon as events up to and including a given choice are fully replicated across the swarm, all machines will agree upon which branch is taken (i.e. we achieve *eventual consensus* [38]).

The fact that events are processed only by subscribers makes the resolution of choices subtle. Assume that the accounting office incorrectly subscribes to the cancellation event but not to the arrival one. In the presence of a conflicting choice, it may incorrectly conclude that the ride was cancelled even when all other roles understand that the ride took place. We rely on a typing discipline for ruling out such inconsistencies. We follow a top-down approach featuring *swarm protocols*, namely abstractions similar to the diagram in Example 1.1 that – akin to global types [19, 20] – formalise a description of the expected protocol from a *global* viewpoint. A projection operation can automatically generate local specifications of each role formally defined as *machines* (cf. Section 2.3). Our typing discipline establishes sufficient conditions – *well-formedness* of swarm protocols – to guarantee that well-typed systems will resolve conflicting choices consistently once information has sufficiently spread to participants. This and the fact that swarm protocols fully abstract away from the number of instances enacting a role are distinguished features of our approach.

Main contributions. We develop a behavioural typing discipline for local-first software tailored to a formal operational model distilled from a real middleware. More specifically:

1. We introduce an operational model for distributed computation based on replication of event logs to drive the behaviour of machines (Section 2). We do not assume log stability but combine speculative computation with a “rewind” mechanism à la *time warp machine* [22]: a conflict is resolved by backtracking and re-execution along the right path.
2. We define a novel behavioural type approach (Section 4) in which swarm protocols are specified in terms of the information injected into a heterogeneous swarm through the actions performed by participants of specific roles. Swarm protocols enjoy a lightweight syntax and simple operational semantics; they are deadlock-free and communication-safe; yet they are expressive enough for modelling complex protocols.
3. We define well-formedness conditions for swarm protocols (Definition 6.7) and a projection operation to derive local machine specifications (Section 5). These ensure eventual consistency between local observations and globally specified behaviour (Theorem 7.14 and Corollary 7.15), which is non-trivial due to the absence of any infrastructure coordination, non-homogeneous event subscriptions across roles, and the ability to implement a role with an arbitrary positive number of replicas.
4. We apply our approach to the TypeScript language and Actyx middleware in the form of a runtime library and a tool for checking protocol well-formedness and conformance, as well as a stochastic simulation tool exploring possible executions.

Assumptions. We work under the following assumptions (cf. Section 9 for extensions):

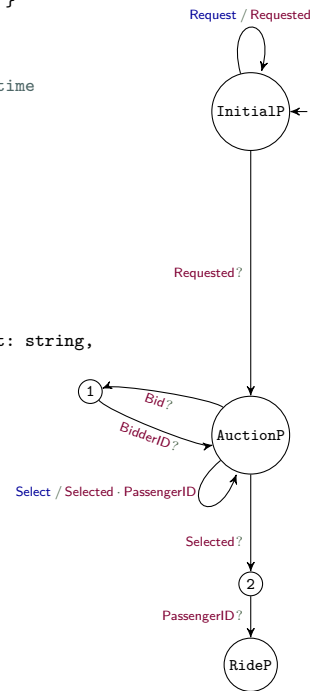
- *collaborative setting*: we consider P2P participants to not be malicious or adversarial;
- *pure effects*: effects performed by machines can be reverted or compensated;
- *reliable pub-sub mechanism*: events (incl. metadata) are neither forged nor lost;
- *session encoding*: machines only receive events pertaining to their session.

Structure. We present the semantics of machines and of log shipping in Section 2. An overview of the accompanying tooling is in Section 3. Swarm protocols are introduced in Section 4 and their local projection onto machines in Section 5. We define well-formed


```

1 // analogous for other events; "type" property matches type name (checked by tool)
2 type Requested = { type: 'Requested'; pickup: string; dest: string }
3 type Events = Requested | Bid | BidderID | Selected | ...
4
5 /** Initial state for role P */
6 @proto('taxiRide') // decorator injects inferred protocol into runtime
7 export class InitialP extends State<Events> {
8   constructor(public id: string) { super() }
9   execRequest(pickup: string, dest: string) {
10    return this.events({ type: 'Requested', pickup, dest })
11  }
12  onRequested(ev: Requested) {
13    return new AuctionP(this.id, ev.pickup, ev.dest, [])
14  }
15 }
16 @proto('taxiRide')
17 export class AuctionP extends State<Events> {
18   constructor(public id: string, public pickup: string, public dest: string,
19     public bids: BidData[]) { super() }
20   onBid(ev1: Bid, ev2: BidderID) {
21     const [ price, time ] = ev1
22     this.bids.push({ price, time, bidderID: ev2.id })
23     return this
24   }
25   execSelect(taxiId: string) {
26     return this.events({ type: 'Selected', taxiID },
27       { type: 'PassengerID', id: this.id })
28   }
29   onSelected(ev: Selected, id: PassengerID) {
30     return new RideP(this.id, ev.taxiID)
31   }
32 }
33 @proto('taxiRide')
34 export class RideP extends State<Events> { ... }

```



■ Listing 1 Definition of state machines in TypeScript.

swarm protocols in Section 6 and study eventual correctness in Section 7. Related works are discussed in Section 8 and Section 9 yields final remarks together with possible generalisations and future work.

2 Asymmetric Replicated State Machines

Our model hinges on three ingredients: machines, event emission and consumption, and logging. The behaviour of a machine is captured by a finite-state automaton as described in Section 2.1. In Section 2.2 we show how machines may offer *commands* that, upon execution, emit events as well as how machines consume (typed) events stored in their *local log*. Such events are immediately stored in the local log of the emitting machine and later asynchronously shipped to the other machines as described in Section 2.3.

2.1 From TypeScript to automata

We formalise (and elaborate on) the computational model realised in the middleware of Actyx by offering a new library for writing endpoint code. Like the Actyx SDK, we use the TypeScript language. Our API focuses on a concise but well-structured expression of finite-state machines for interpreting the current state of distributed computation.

We illustrate this by considering the implementation of the request and auction part of our running taxi example from the passenger's point of view, with the TypeScript code given in Listing 1. For the purposes of this section – all details, including runtime evaluation, are given in Section 3 – it suffices to know that each state of a machine is represented by a

■ **Table 1** Notation for machines.

Notation:	State computation:
$\vdash e : \mathbf{t}$: event e is of type \mathbf{t} $src(e)$: identity of the machine generating e l : event log (seq. without repetition) \mathbf{l} : event log type (sequence of types) \mathbf{c} / \mathbf{l} : command \mathbf{c} emits log type \mathbf{l} $\mathbf{t}?$: consumption of event of type \mathbf{t} \mathbf{M} : machine (labelled transition system)	Let q_0 be the initial state of \mathbf{M} and $\mathbf{M}[q]$ be machine \mathbf{M} with initial state changed to q : $\delta(\mathbf{M}, \epsilon) = q_0$ $\delta(\mathbf{M}, e \cdot l) = \begin{cases} \delta(\mathbf{M}[q], l) & \text{if } \vdash e : \mathbf{t}, q_0 \xrightarrow{\mathbf{t}^?} q \text{ in } \mathbf{M} \\ \delta(\mathbf{M}, l) & \text{otherwise} \end{cases}$

TypeScript class, with methods for command invocation whose name is prefixed with “**exec**”, and event handler methods to compute the next state (names prefixed with “**on**”). The types of the event handler method arguments are significant, as are the return types of command methods. Listing 1 also depicts the finite-state automaton corresponding to the snippet, as inferred by the `machine-check` build tool (cf. Section 3.2):

- states `InitialP`, `AuctionP`, and `RideP` of the automaton respectively correspond to the classes in the snippet with the same name;
- states 1 and 2 of the automaton correspond to the implicit states interspersed between the events specified as arguments to the event handler methods `onBid` and `onSelected`;
- command methods correspond to self-loops in the automaton, labelled with the command name and the resulting event log type, such as `Request / Requested` in state `InitialP`;
- event handlers correspond to transitions or sequences thereof, where each transition is labelled with an event type, such as `Requested?` from state `InitialP`.

The correspondence sketched above is the basis for our formalisation of swarms and it is at the heart of the library `machine-runner` introduced in Section 3.

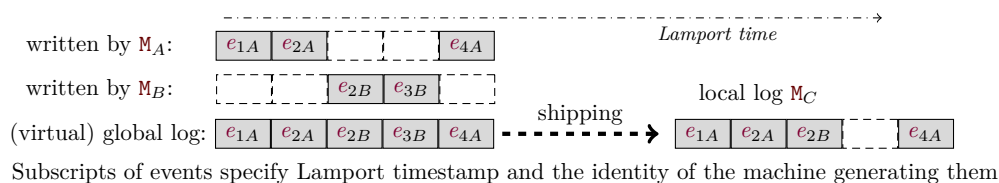
Note that in the automaton we abstract away from payloads, considering only the types of events. We also ignore internal computations not involving event emission/consumption (e.g. the computation of the constructor arguments for state `AuctionP` is immaterial).

2.2 Commands execution and events consumption

A machine can be thought of as the proxy of an agent (algorithm or human) that processes the information in the local log, comes to conclusions, and makes decisions which may lead to the invocation of an enabled command. For instance, in state `AuctionP`, the machine in Section 2.1 enables the passenger to execute the command `Select` triggering the emission of a log like `selected · passengerid`. This sequence of events is added to the local event log making the machine move to state 2 first by consuming the event `selected` and then to state `RideP` by consuming the event `passengerid`. This is why the inferred machine state diagram records commands as self-loops while only event consumption may induce state changes.

The automata representation of machines discussed in Section 2.1 allows us to limit technicalities in defining the behaviour of machines. We illustrate the notation given in Table 1 by abstracting the above example: a machine \mathbf{M} enabling command \mathbf{c} will upon invoking that command emit a sequence of events $e_1 \cdot e_2 \cdot \dots = l$ (called a log); the events are decidable typed as $\vdash e_i : \mathbf{t}_i$, with $\mathbf{t}_1 \cdot \mathbf{t}_2 \cdot \dots = \mathbf{l}$ being the log type associated with \mathbf{c} . We only consider deterministic machines, i.e. the labels of event transitions $\mathbf{t}?$ are pairwise distinct.

For the purpose of enabling commands a machine \mathbf{M} with log l is implicitly in a state denoted $\delta(\mathbf{M}, l)$. The determinism of \mathbf{M} ensures that there is a unique such state. $\delta(\mathbf{M}, l)$ is a *transition function* defined by adapting the standard transition function of finite-state automata. Starting with the initial state of \mathbf{M} we inductively remove the oldest event, say e ,



■ **Figure 1** Events sliced by their source: each event starts out at the machine where it is emitted. Logs are disseminated such that the recipient (like machine M_C) holds a prefix of each of the source slices, which is a partial view of the global log. The recipient’s local log is ordered like the global log. Eventually every event arrives at all machines, filling the transient gaps that may have existed.

and check it against the outgoing transitions of the current state: if M has a transition with label t ? and e has type t we transition to its target state, otherwise the event e is dropped; in either case we repeat this step until the log is empty.

It is important to note that a command may be invoked only if it is enabled in the state reached after fully processing the local log. Also, emitted events are *appended* to the local log of the machine they originate at. This ensures causality preservation since the new events are ordered *after* all events previously known by this machine.

2.3 Swarms and log-shipping

The last piece of our computational model is the mechanism for disseminating event logs among the machines of a swarm. This mechanism affects the behaviour of a recipient: as described in Section 2.2, the local log contains more events, leading to a new current state being computed, which in turn may change the set of available commands.

Our goal is eventual consensus between machines, in particular different replicas of the same machine must reach the same state when consuming the same events. According to the definition of the state transition function in Table 1, this can be achieved only if the events are ordered in the same way in the local logs. We address this by enforcing a total order between events, without requiring coordination between machines. As discussed in the previous section this total order preserves causality. Note that the ordering of events that are not yet locally known is arbitrary but well-defined, and to capture this concept we introduce the notion of a global log. Figure 1 illustrates the dissemination of logs, where each event begins in the local log of the emitting machine and simultaneously takes its place in the virtual global log based on the total order. We model log-shipping as a machine enlarging its local log with events from the global log; in practice, events flow from the local log of a machine to another machine’s local log. The precise algorithm for selecting the source and destination is not relevant to our theory. Due to the uncoordinated total order, it may happen that an incoming event is sorted into the middle of a local log, which can alter the interpretation of all subsequent events and affect the computation of the current state.

For example, consider the case in our running example in which the passenger selects the taxi at the same time that another taxi places a new bid. If the passenger’s selection is ordered before the bid, a later inspection of the log may reveal that the passenger selected suboptimally, but the selection still remains in effect and the bid is ignored. (In a system based on a central database the “bid” transaction would be rejected instead.) On the other hand, if the event *selected* were placed in between *bid* and *bidderid*, it would be ignored once the logs are replicated. In this case there are two reasonable paths forward: honouring the passenger’s previous wish would require a *compensating action* of executing the same selection again, or the selection could be redone including the new bid, possibly leading to a different outcome – this workflow choice needs to be made by the application designer.

■ **Table 2** Swarm semantics: coinductively unfold machines to inductively build up logs.

<p>Notation:</p> <p>\mathbf{S} : a list of pairs (\mathbf{M}_i, l_i)</p> <p>(\mathbf{S}, l) : a system (machines with local logs paired with a global log)</p> <p>κ : set of command invocations $\mathbf{c} / \mathbf{1}$</p> <p>$\implies = \longrightarrow^*$ where $\longrightarrow = \xrightarrow{\tau} \cup \bigcup_{\mathbf{c}, \mathbf{1}} \xrightarrow{\mathbf{c} / \mathbf{1}}$</p>	<p>Log merging:</p> <p>Let \sqsubseteq be the sublog relation of Definition 2.1:</p> $l_1 \bowtie l_2 = \{l \mid l \subseteq l_1 \cup l_2 \text{ and } l_1 \sqsubseteq l \text{ and } l_2 \sqsubseteq l\}$ <p>Machine step:</p> <p>If $\delta(\mathbf{M}, l)$ has a self-loop with label $\mathbf{c} / \mathbf{1}$ and $\vdash l' : \mathbf{1}$ then $(\mathbf{M}, l) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{M}, l \cdot l')$.</p>
<p>Operational semantics:</p> $\frac{\mathbf{S}(i) = (\mathbf{M}, l_i) \quad (\mathbf{M}, l_i) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{M}, l'_i) \quad \text{src}(l'_i \setminus l_i) = \{i\} \quad l' \in l \bowtie l'_i}{(\mathbf{S}, l) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{S}[i \mapsto (\mathbf{M}, l'_i)], l')} \text{[LOCAL]}$ $\frac{\mathbf{S}(i) = (\mathbf{M}, l_i) \quad l_i \sqsubseteq l'_i \sqsubseteq l \quad l_i \subset l'_i}{(\mathbf{S}, l) \xrightarrow{\tau} (\mathbf{S}[i \mapsto (\mathbf{M}, l'_i)], l)} \text{[PROP]}$	

2.4 Formalisation

A *swarm* (of size n) is a pair (\mathbf{S}, l) where \mathbf{S} maps indices $1 \leq i \leq n$ to machines and their local log, i.e. $\mathbf{S}(i) = (\mathbf{M}_i, l_i)$ and l is the global log. It is convenient to let $(\mathbf{M}_1, l_1) \mid \dots \mid (\mathbf{M}_n, l_n) \mid l$ denote the swarm (\mathbf{S}, l) such that $\mathbf{S}(i) = (\mathbf{M}_i, l_i)$ for $1 \leq i \leq n$. A swarm $(\mathbf{M}_1, l_1) \mid \dots \mid (\mathbf{M}_n, l_n) \mid l$ is *coherent* when the local log l_i of each machine \mathbf{M}_i is made of events actually emitted in \mathbf{S} in the order in which they appear in the global log.

► **Definition 2.1** (Sublogs and coherence). *A log $l = e_1 \dots e_n$ induces a total order $<_l$ on its elements as follows: $e_i <_l e_j \iff i < j$. The sublog relation on logs \sqsubseteq demands an order-preserving and downward-complete morphism from l_1 into l_2 . Formally, $l_1 \sqsubseteq l_2$ if*

1. *all events of l_1 appear in l_2 ($l_1 \subseteq l_2$) in the same order ($<_{l_1} \subseteq <_{l_2}$); and*
2. *the per-source partitions of l_1 are prefixes of the corresponding partitions of l_2 , i.e. for all $e_1 \in l_1, e_2 \in l_2$ from a given src , $e_2 <_{l_2} e_1$ implies $e_2 \in l_1$.*

A swarm $(\mathbf{M}_1, l_1) \mid \dots \mid (\mathbf{M}_n, l_n) \mid l$ is coherent if $\bigcup_{1 \leq i \leq n} l_i = l$ and $l_i \sqsubseteq l$ for $1 \leq i \leq n$.

The operational semantics of swarms accounts for the construction of the global log, i.e. the total order defined over generated events. To model that this assignment is non-deterministic we rely on the merge operator $_ \bowtie _$ defined in Table 2 to combine two logs that may share events. This operator generates all possible logs that contain the events from both original logs while maintaining their input order.

The operational semantics is given by the rules LOCAL and PROP in Table 2. They respectively formalise the effects of command execution described in Section 2.2 and the non-deterministic log-shipping mechanism illustrated in Section 2.3. Recall that a machine \mathbf{M} with a log l is implicitly in state $\delta(\mathbf{M}, l)$. We can hence define the relation $(\mathbf{M}, l) \xrightarrow{\mathbf{c} / \mathbf{1}} (\mathbf{M}, l \cdot l')$ holding when the state $\delta(\mathbf{M}, l)$ enables the command $\mathbf{c} / \mathbf{1}$ and l' has type $\mathbf{1}$. The type of a log $\mathbf{1}$ is the sequence of the types of its events. We write $\vdash l : \mathbf{1}$ when l has type $\mathbf{1}$; this is decidable since the typing of events (cf. Table 1) is decidable.

Rule LOCAL describes the invocation of the command \mathbf{c} enabled at the i -th machine of the swarm (\mathbf{S}, l) . In addition to updating the local log of the i -th machine to l'_i , which extends l_i with the events generated by \mathbf{c} , the rule also updates the global log. The new global log now

includes the events generated by the i -th machine, assigning their place in the total order by picking one of the possible orders generated by the merge operator defined in Table 2. Rule PROP defines event log propagation between machines. The idea is to non-deterministically select a machine whose local log is a strict sublog of the global log, identify a larger sublog $l'_i \sqsubseteq l$, and transfer events to the machine by assigning l'_i as its new local log.

Note that our formalisation acts on the logs which grow by appending newly generated events. These features play an important role in the realisation of the local-first principle and permit to formally represent the conflicts discussed in Section 1.

Let \Longrightarrow be the reflexive and transitive closure of the operational semantics relation (cf. Table 2). The following properties hold on coherent swarms.

- **Lemma 2.2** (Coherence preservation and eventual consistency). *Given a coherent swarm $(S, l) = (M_1, l_1) \mid \dots \mid (M_n, l_n) \mid l$ then*
- coherence preservation: $(S, l) \xrightarrow{\alpha} (S', l')$ implies that (S', l') is coherent
 - eventual consistency: $(S, l) \Longrightarrow (M_1, l) \mid \dots \mid (M_n, l) \mid l$.

3 Tool support

Our theoretical development is accompanied by a set of software tools that support the implementation of swarms as a composition of type-checked TypeScript machines [30] and runs them based on the Actyx middleware [1]. The ecosystem is depicted in Figure 2.

3.1 Execution of compiled machines

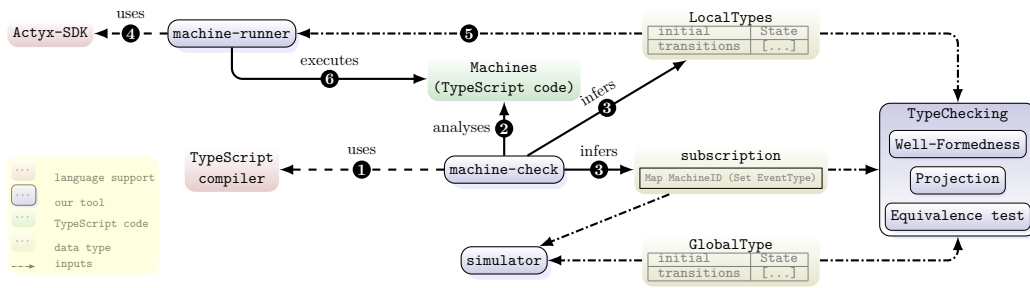
The `machine-runner` library uses the Actyx SDK (cf. arrow ④ in Figure 2) to drive machines written in TypeScript; more precisely, it employs ⑤ local types to interpret incoming events and execute ⑥ the corresponding machine logic. The declaration of a machine revolves around the event types that it can handle. Referring back to Listing 1, we show the `Requested` event type on line 2 as an example: using a property called `type` to hold a string of singleton type (here: `'Requested'`) is a customary way to express a *tagged union* in TypeScript, as shown on line 3.

Every machine state is represented by a class that derives from the `State` base class (or *prototype* in JavaScript terms) provided by the `machine-runner` library. This serves both as a marker for machine states and to carry the type parameter constraining all emitted events to a common type: the inherited `this.events` function used for example on line 28 is a utility for helping TypeScript to correctly capture the tuple type `[Selected, PassengerID]` (instead of the otherwise inferred array type `Events[]`) and assert that each of the event types conforms to type `Events`.

A program using the passenger’s machine would start by constructing the initial state using for example `new InitialP('myID')`. Together with a suitable set of Actyx event tags (like `'ride:12345'` to tag this particular taxi ride protocol session) and a state change callback (see below). This initial state is then passed to the library’s `runMachine` function. This will set up a subscription for events with those tags using the Actyx SDK, where Actyx will first deliver all historic events already locally known and then switch to live mode.

Whenever an event is received, it is slated for consumption by the appropriate event handler method; to this end the handler method for the event’s type is dynamically looked up in the JavaScript object underlying the state’s class. If that method takes only a single argument then the event is immediately consumed by calling the method, which returns the next state of the machine. Otherwise, the event is enqueued, awaiting the receipt of the following required event type etc. until the desired event sequence is complete and

15:10 Behavioural Types for Local-First Software



■ **Figure 2** Tool ecosystem.

the method can be invoked with all arguments. During this whole process, whenever the incoming event type does not have a matching handler or is not of the next required type in an argument list, the event is discarded as detailed in Section 2.2.

Whenever the machine state changes (i.e. when $\delta(M, l)$ computes a new value), the new state is passed to a function that the application passed to `runMachine` earlier – this scheme is termed a *callback* in TypeScript (note that this language implements an imperative style with mutable bindings). This could update a user interface or trigger an algorithm to compute reactions. The state’s command methods can therein be used to construct adequate event payloads for enabled commands, which would then be stored in Actyx using an SDK function and come back via the event subscriptions – now with metadata – to be applied to the current state and eventually trigger another invocation of the callback.

3.2 Enforcing typing at run-time

Readers versed in TypeScript may have noticed that we glossed over a difficulty here: TypeScript types are fully erased at runtime, meaning that the `machine-runner` code will not be able to find the event handler method by using the event type, and it will also not know how many arguments that method takes and what its types are. Therefore, the first responsibility of the `machine-check` build tool is to analyse ② the TypeScript code and ascertain that all event types are declared such that they can be recognised at runtime on their `type` property – the handler method’s name can then be constructed by prefixing the value of this property with `'on'`. The second responsibility is to extract the function signatures of all event handlers, check that each handler’s name corresponds to the name of its first argument type, and then construct a per-state mapping from first event type to the list of following events (possibly empty). This information is made available to `runMachine` by decorating [37] the user-written state class: the `@proto` decorator transforms the class definition as it is loaded by the JavaScript VM, overriding the `reactions` method inherited from the `State` prototype. To do that, the implementation of the `proto` function needs to access the `machine-check`’s extraction results. This is done by importing ⑤ a source module generated by `machine-check` that contains all protocol information in JSON format [36].

While the aforementioned duties of `machine-check` are crucial for `machine-runner`’s operation, the more interesting function of this build tool relates to the inference of local types and subscriptions (arrows ③ in Figure 2) as well as initiating the type-checking process on swarm protocols. To this end, the TypeScript compiler is used ① as a library to obtain ② a fully typed AST representation of the user program. Since we are only interested in machines, our entry points are `State` subclasses that are marked as initial states by a documentation

comment starting with “Initial state for role”, as is shown on line 5 of Listing 1. This comment serves the secondary purpose of naming the role this machine aspires to play (P for passenger in this example). The `@proto` decorator on line 6 not only has its runtime duties as explained above, it also carries in its argument the name of the swarm protocol that provides the context for the role name – we discuss both concepts in detail in the following sections; `machine-check` expects to find the definition of the swarm protocol in a correspondingly named file in JSON format. Finally, `machine-check` assembles the lists of command and event handler methods by inspecting (arrow ⑤) a state class’s method names and signatures and follows up with recursively processing the result types of event handlers in the same fashion. Any event type seen in an event handler argument list is automatically added to the subscription set of the machine (needed for the projection as explained in Section 5).

3.3 Type-checking, simulation and more

As a result of the analysis described above `machine-check` has assembled the following pieces for each machine definition within the user program: swarm protocol, role name, subscriptions, states, and transitions. Each such tuple is then passed – again in JSON format – to the `typechecking` tool, our third artifact contribution, written in Haskell. This tool first checks that the provided swarm protocol and subscription are well-formed (according to the rules presented in Section 6), computes the projection for the given role, and finally checks the inferred machine type for equivalence to the projection result (where state names are immaterial).

We provide a tool written in Haskell for the simulation of the formal operational semantics of the model. For a given protocol and subscription, the tool computes the projections and simulates the execution of swarms consisting of machines according to those projections. It supports both exhaustive and random generation of traces up to a given length. The tool has been used for checking claims and results about our running example.

The aforementioned tools are detailed in [28]. Through an example project, the accompanying paper also demonstrates the use of the inferred machine type to generically render a machine UI. Besides showing the current state of the computation, the UI gives the user the possibility to interact with machines by invoking enabled commands, where command arguments are gathered using automatically generated HTML forms.

4 Swarm protocols

A *swarm protocol* (hereafter also called *protocol* for short) describes the intended overarching event log structure realised by a swarm of machines; it corresponds to a global type in the terminology of session types, with our machines playing a similar role to local types. The protocol captures the overall communication structure as well as the details relevant for implementing it with machines. As it is customary with behavioural types, swarm protocols rely on an idealised environment where all communication is infallible and instantaneous. The link to the realisation in terms of machines is given in the following section by way of a projection operation.

When defining the syntax of swarm protocols, we follow the approach initiated in [7] that avoids fixing a syntactic representation of recursion and simplifies later treatment by instead using infinite regular trees. A swarm protocols is a possible infinite, *regular* term coinductively generated by the grammar in Table 3. A term is regular if it consists of finitely many *distinct* subterms. The language generated by the coinductive grammar is thus finitely representable either using the so-called “ μ notation” [31] or as solutions of finite sets of equations [9]. The

■ **Table 3** Swarm protocols: traverse a coinductive type to inductively build up an event log.

<p>Swarm protocol syntax: Regular terms generated by:</p> $\mathbf{G} ::= \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{l}_i \rangle . \mathbf{G}_i \mid \mathbf{0}$ <p>\mathbf{R}_i : role $\mathbf{c}, \mathbf{l}, l$: as for machines</p>	<p>State computation:</p> $\mathbf{G} \xrightarrow{\mathbf{c}_j / \mathbf{l}_j} \mathbf{G}_j \iff \mathbf{G} = \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{l}_i \rangle . \mathbf{G}_i \text{ and } j \in I$ $\delta(\mathbf{G}, l) = \begin{cases} \mathbf{G} & \text{if } l = \epsilon \\ \delta(\mathbf{G}', l'') & \text{if } \mathbf{G} \xrightarrow{\mathbf{c}/\mathbf{l}} \mathbf{G}' \text{ and } \vdash l' : \mathbf{l} \text{ and } l = l' \cdot l'' \\ \perp & \text{otherwise} \end{cases}$
<p>A swarm protocol is well-formed (Def. 6.7) if it is causal consistent (Def. 6.1), choice determinate (Def. 6.3), and confusion-free (Def. 6.5).</p>	
<p>Operational semantics:</p>	$\frac{\delta(\mathbf{G}, l) \xrightarrow{\mathbf{c}/\mathbf{l}} \mathbf{G}' \quad \vdash l' : \mathbf{l} \quad l' \text{ log of fresh events}}{(\mathbf{G}, l) \xrightarrow{\mathbf{c}/\mathbf{l}} (\mathbf{G}, l \cdot l')} \text{[G-CMD]}$

interested reader is referred to [9] for a comprehensive treatment. Intuitively, the protocol progresses by some role \mathbf{R}_i invoking command \mathbf{c}_i , appending a non-empty event sequence of type \mathbf{l}_i to the global log and continuing as protocol \mathbf{G}_i . As discussed at the end of Section 2.4, the resolution of the choice specified in a swarm protocol is not coordinated among the instances of the roles involved in the choice (i.e. there is no unique selector). In fact, instances of different roles involved in the choice may enable commands at the same time as well as different instances of the same role may enable different commands (recall that each machine tracks a separate local log and event replication is asynchronous). This is in contrast to most other behavioural type systems hitherto, which do not permit such race conditions.

► **Definition 4.1** (Determinism). *A protocol $\mathbf{G} = \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{l}_i \rangle . \mathbf{G}_i$ is log-deterministic if the event types $\mathbf{l}_i[0]$ are pairwise different and all \mathbf{G}_i are log-deterministic. \mathbf{G} is command-deterministic if the tuples $(\mathbf{c}_i, \mathbf{R}_i)$ are pairwise different and all \mathbf{G}_i are command-deterministic. \mathbf{G} is deterministic if it is log-deterministic and command-deterministic.*

Hereafter we only consider deterministic swarm protocols. Note that determinism is evidently decidable on swarm protocols due to the regularity constraint.

► **Example 4.2** (Taxi service). The swarm protocol for the scenario in Example 1.1 is

$$\begin{aligned} \mathbf{G} &= \text{Request}@P \langle \text{Requested} \rangle . \mathbf{G}_{\text{auction}} \\ \mathbf{G}_{\text{auction}} &= \text{Offer}@T \langle \text{Bid} \cdot \text{BidderID} \rangle . \mathbf{G}_{\text{auction}} + \text{Select}@P \langle \text{Selected} \cdot \text{PassengerID} \rangle . \mathbf{G}_{\text{choose}} \\ \mathbf{G}_{\text{choose}} &= \text{Arrive}@T \langle \text{Arrived} \rangle . \text{Start}@P \langle \text{Started} \rangle . \mathbf{G}_{\text{ride}} + \text{Cancel}@P \langle \text{Cancelled} \rangle . \text{Receipt}@O \langle \text{Receipt} \rangle . \mathbf{0} \\ \mathbf{G}_{\text{ride}} &= \text{Record}@T \langle \text{Path} \rangle . \mathbf{G}_{\text{ride}} + \text{Finish}@P \langle \text{Finished} \cdot \text{Rating} \rangle . \text{Receipt}@O \langle \text{Receipt} \rangle . \mathbf{0} \end{aligned}$$

The structure in terms of commands and roles is straightforwardly induced from Example 1.1, with event log types filled in according to further requirements. The event type **BidderID** represents identifying information illustrating that not all events are of interest to all roles (e.g., the office does not need to know all bidders, it only needs to know which taxi was **Selected**). It is straightforward to check that \mathbf{G} is both log- and command-deterministic. ◻

Mirroring the formulation of machines we ascribe operational semantics to a swarm protocol via the generation and processing of an event log. The main difference in state computation is that swarm protocols generate and consume logs instead of events, as illustrated with the **Offer**, **Select**, and **Finish** commands in the example above.

The state reached by a swarm protocol \mathbf{G} after processing a log l is computed using an extension of the transition function δ as defined in Table 3. Analogously to the definition for machines, the transition function $\delta(\mathbf{G}, l)$ returns the continuation of the swarm protocol \mathbf{G} after processing the entire log l . We stress that δ is a partial function on swarm protocols, it is undefined when log l cannot be generated according to \mathbf{G} (unlike the definition of δ on machines, which is a total function since it just discharges unrecognised events). Note also that δ is well-defined over log-deterministic swarm protocols because a log in a branch cannot appear as a prefix of the logs of the remaining branches of the choice.

The operational semantics of a swarm protocol is defined as a labelled transition system given by rule [G-CMD] in Table 3. This rule states that a swarm protocol \mathbf{G} with log l enables command c , upon whose invocation the log is extended with fresh events l' of type $\mathbf{1}$ before possibly allowing another command to be invoked. The freshness of the events in l' can e.g. be guaranteed by the inclusion of node ID and logical timestamp.

► **Example 4.3** (Idealised taxi service). Consider the protocol from Example 4.2, starting out with (\mathbf{G}, ϵ) . After invoking the **Request** command our log contains *requested* and we reach state $\mathbf{G}_{\text{Auction}}$. Two bids later the passenger makes their selection, leading us to

$$\delta(\mathbf{G}, \text{requested} \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{selected} \cdot \text{passengerid}) = \mathbf{G}_{\text{choose}}$$

with $\mathbf{G}_{\text{choose}}$ offering two options: either the passenger invokes **Cancel** or the taxi **Arrives**. ◻

5 Projection

For the definition of our projection operation it is convenient to introduce a textual presentation of machines equivalent to the automata-based presentation used so far. Let κ denote a finite function mapping commands to non-empty log types; we allow ourselves to treat κ as set (the graph of function κ) and e.g. write $c / \mathbf{1} \in \kappa$ for $\kappa(c) = \mathbf{1}$ or else write $\{c_1 / \mathbf{1}_1, \dots, c_h / \mathbf{1}_h\}$ for the function κ mapping c_i to $\mathbf{1}_i$ for each $i \in \{1, \dots, h\}$. t_i ranges over event types.

Similarly to swarm protocols, the textual presentation of our machines is a regular term² of the following coinductive grammar:

$$\mathbf{M} ::=^{\text{co}} \kappa \cdot [t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n] \quad (1)$$

and we abbreviate $\kappa \cdot [t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n]$ as $\kappa \cdot \mathbf{0}$ when $n = 0$ and as $t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n$ when κ is the empty map. We also write $\&_{1 \leq i \leq n} \mathbf{1}_i?M_i$ in place of $t_1?M_1 \ \& \ \dots \ \& \ t_n?M_n$.

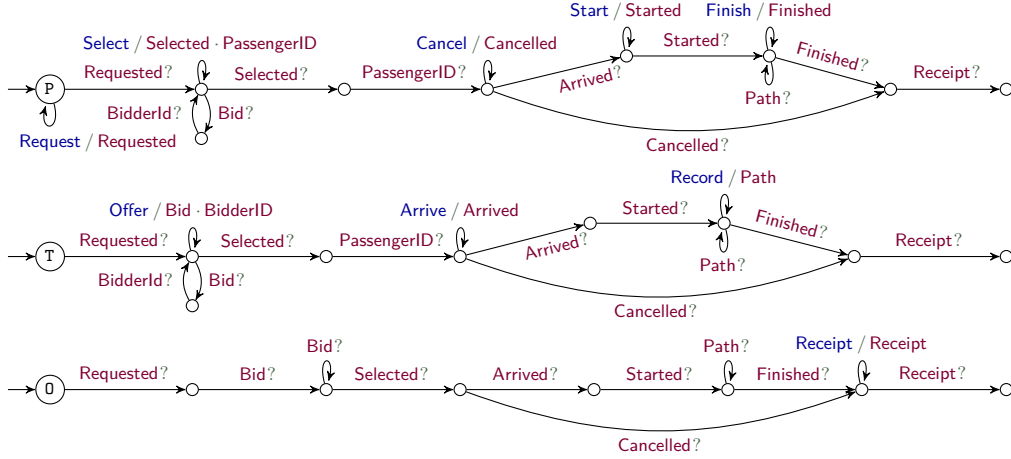
In turning our attention to projection operations we first note that the responsibility for driving the protocol forward is distributed across the participants: each transition in the swarm protocol is labelled with one role that may trigger it by invoking the command. Each machine plays one role, whose machine specification is obtained by the projection operation $\mathbf{G} \downarrow_{\mathbf{R}}$. Note that multiple machines may implement the same role.

One could define $\mathbf{G} \downarrow_{\mathbf{R}}$ such that each transition in \mathbf{G} produces a series of event transitions in the machine plus a command invocation on the originating state if the role matches.

$$\left(\sum_{i \in I} c_i @_{\mathbf{R}_i} \langle \mathbf{1}_i \rangle \cdot \mathbf{G}_i \right) \downarrow_{\mathbf{R}} = \kappa \cdot [\&_{i \in I} \mathbf{1}_i? \mathbf{G}_i \downarrow_{\mathbf{R}}] \quad \text{where} \quad \kappa = \{(c_i / \mathbf{1}_i) \mid \mathbf{R}_i = \mathbf{R} \text{ and } i \in I\}$$

² The correspondence between these regular terms and finite-state automata yields exactly the presentation of machines in terms of finite-state automata that we have adopted so far.

15:14 Behavioural Types for Local-First Software



■ **Figure 3** Projection of Example 4.2 on P, T, and O as automata.

Albeit simple, this projection scheme generates unnecessarily large machines in all but the most trivial cases. More crucially, forcing each machine to process all events is undesirable for reasons of security and efficiency. It would be highly desirable to allow some information to be kept secret from certain roles (like `passengerID` in Example 4.2), and it would be most efficient if every role processed just enough information to correctly enable and disable command invocations. We therefore define a more appealing construction.

Our projections are based on the notion of whether a machine shall process a certain type of event. Formally, the projection operation is parameterised by a *subscription*, namely a map σ assigning to each role the set of event types that it reacts to. Given a set of log types E , let $filter(_, E)$ be a function transforming a log type, retaining only the event types in E while preserving their relative order. Intuitively, subscriptions correspond to topics in a publish–subscribe framework whereby processes declare which kinds of messages they are interested in receiving.

► **Definition 5.1** (Projection). *Given a swarm protocol G and a subscription σ , the projection of G over a role R with respect to σ , written $G \downarrow_R^\sigma$, is defined as follows:*

$$\left(\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle \cdot G_i \right) \downarrow_R^\sigma = \{ c_i / \mathbf{1}_i \mid R_i = R \text{ and } i \in I \} \cdot [\&_{j \in J} filter(\mathbf{1}_j, \sigma(R)) ? (G_j \downarrow_R^\sigma)]$$

where $J = \{ i \in I \mid filter(\mathbf{1}_i, \sigma(R)) \neq \epsilon \}$.

Notice that we omit the projection of a branch when a role R is not subscribed to any of the event types emitted by the command that selects that branch. We opted for this simplification of the formalism because our well-formedness conditions (cf. Section 6) ensure that if a role is involved in the continuation it will subscribe to the the first event in the branch. Further, note that this pruning applies to branches in isolation, later states reachable by other paths remain part of the projection.

► **Example 5.2.** Let σ be such that $\sigma(P)$ consists of all the event types in the protocol G defined in Example 4.2. The projections of G are in Figure 3. ┘

6 Well-formedness

We now focus on the *well-formedness* conditions of our swarm protocols. As is standard in behavioural types, sufficient conditions are established on global specifications that guarantee relevant properties on projections such as deadlock or lock freedom and absence of orphan messages. The properties of interest to us are quite different from those common in standard settings since we aim to guarantee that eventual consensus is reached even when some of the participants make choices that are discordant due to their incomplete view on the global log. The idea is that transitory deviations are *tolerated* provided that consistency is eventually recovered, which happens once information has sufficiently spread within the swarm. For instance, a taxi in our running example may keep bidding for a passenger's auction after the passenger has made their selection as long as the selection event has not yet been received. This temporary inconsistency is recognised and resolved once the events have propagated to the deviating taxi and the passenger, respectively.

Realising swarms with this property is not straightforward. The rest of this section illustrates the problems arising in our setting with a few examples. For each problem we identify sufficient conditions on our swarm protocols that rule out the problem for coordination issues in realistic scenarios based on our running example). These conditions culminate in our definition of *well-formedness* (cf. Definition 6.7).

6.1 On causality and propagation

The first problem we look at is related to how a command is disabled once it has been invoked. In our setting, this boils down to fine tuning the registration of roles to event types. For example, if a command c should be enabled only after another, say c' has been executed, then the role executing c' should be subscribed to some event type emitted in response to the execution of c . Another example is that a command can stay perpetually enabled if the role executing it is oblivious of all resulting events (cf. [27]).

Another class of problems is caused by the fact that events propagate asynchronously within a swarm and that an emission of multiple events is not guaranteed to reach all other machines as one atomic transmission (cf. [27]) This anomaly may be excluded by a runtime system that never applies the [PROP] rule to a strict subset of the event log emitted by a single command, i.e. it treats the log from each command invocation as an atomic unit. We chose to not restrict the way in which a runtime system should propagate events between network sites because we consider it important that implementations be free to optimise their strategy in different ways (e.g. for latency, bandwidth, efficiency, or consistency).

We define the *active* roles of a swarm protocol G as those that can select one of the branches in the top-level choice of G and – given a subscription σ – the *roles* of G as those that can invoke commands or are subscribed to events in G . Formally,

$$\begin{aligned} \text{active}\left(\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i\right) &= \{R_i \mid i \in I\} \\ \text{roles}\left(\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i, \sigma\right) &= \bigcup_{i \in I} (\{R \in \text{dom } \sigma \mid R_i = R \vee \mathbf{1}_i \cap \sigma(R) \neq \emptyset\} \cup \text{roles}(G_i, \sigma)) \end{aligned}$$

(note that the latter is a coinductive definition). With this notation we define the following sufficient condition for avoiding the aforementioned problems.

► **Definition 6.1** (Causal consistency). *A swarm protocol $\sum_{i \in I} c_i @ R_i \langle \mathbf{1}_i \rangle . G_i$ is causal-consistent in a subscription σ if for all $i \in I$*

1. $\mathbf{1}_i \cap \sigma(\mathbf{R}_i) \neq \emptyset$, and
2. $\mathbf{R} \in \text{active}(\mathbf{G}_i)$ implies $\mathbf{1}_i \cap \sigma(\mathbf{R}) \neq \emptyset$ and for all $\mathbf{R}' \in \text{roles}(\mathbf{G}_i, \sigma)$, $\mathbf{1}_i \cap \sigma(\mathbf{R}') \subseteq \mathbf{1}_i \cap \sigma(\mathbf{R})$

Condition (1) requires that the role that performs one of the commands c_i should observe some of the corresponding emitted events $\mathbf{1}_i$. This simple mechanism ensures that repeated command invocation can only occur where foreseen in the swarm protocol. Condition (2) ensures the adequate tracking of causality for subsequent command invocations. The first part ensures that the immediately following command must wait for the enabling transition to occur, while the second part guarantees the ordering of the subsequent command's generated events after all events from the preceding command that are observed by some role in the further evolution of the protocol.

► **Example 6.2.** The protocol of the running example \mathbf{G} in Example 4.2 is causal-consistent for the subscription σ in Example 5.2. In fact, the commands generate logs that start with pairwise-different event types. Hence, the conditions straightforwardly hold for roles P and T, which observe every event. For O, we observe that they only execute the command `receipt`; which should be performed after `Cancelled` or `Finished`, which are also observed by O. ─

6.2 On distributed choices

The next anomaly we study is caused by the fact that our model permits multiple roles to be active at the same time without coordination – this property is essential for perfect availability as demanded by local-first cooperation. Such behaviour would be ruled out in all the global type systems we are aware of. Our strategy for coping with the inevitably arising conflicts is that we permit machines to make inconsistent local decisions but reconcile those once the corresponding events have propagated to all relevant parties (e.g., the office in our example can make a choice inconsistent with the decisions taken by other participants as shown in see [27]). We fix this by requiring *determinacy*.

► **Definition 6.3** (Determinacy). A swarm protocol $\mathbf{G} = \sum_{i \in I} c_i @ \mathbf{R}_i \langle \mathbf{1}_i \rangle$. \mathbf{G}_i is determinate for subscription σ if it is causal-consistent and $\mathbf{R} \in \text{roles}(\mathbf{G}_i, \sigma)$ implies $\mathbf{1}_i[0] \in \sigma(\mathbf{R})$ for $i \in I$.

This definition of determinacy is prompted by our determinism rule (Definition 4.1): we identify a branch by the first event type of its emitted log. Note that a role involved in one branch but not in another may invoke commands that are later invalidated without that role being able to recognise this situation; we will explore mechanisms for compensating such errors in future work, which may require strengthening the rule above.

► **Example 6.4.** The swarm protocol \mathbf{G} in Example 4.2 is determinate for the subscription σ in Example 5.2. ─

6.3 On interference

Events emitted by the losing parties to a conflict should be ignored in order to let every machine eventually agree on each choice. Each machine must locally be able to ignore such events, which means that it would be problematic to confuse a machine by emitting such a branch-choosing event in another context (e.g. while proceeding along a sibling branch which this machine does not follow). We avoid this confusion by requiring that any branch of a swarm protocol is communicated using a dedicated event type, i.e. that event type cannot be emitted by any other command. We formulate this notion in terms of the set *subterms*(\mathbf{G}) of all subterms (incl. indirect) of a swarm protocol \mathbf{G} . Recall that this set is finite because our swarm protocols are regular.

In what follows we write $events(\mathbf{G})$ and $guards(\mathbf{G})$ respectively for the sets of all event types and the ones that identify branches; formally, if $\mathbf{G} = \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{1}_i \rangle . \mathbf{G}_i$ then

$$events(\mathbf{G}) = \bigcup_{i \in I} \left(events(\mathbf{G}_i) \cup \bigcup_j \mathbf{1}_i[j] \right) \quad \text{and} \quad guards(\mathbf{G}) = \bigcup_{i \in I} (\{\mathbf{1}_i[0]\} \cup guards(\mathbf{G}_i))$$

(observe that $events(\mathbf{0}) = guards(\mathbf{0}) = \emptyset$ and that these coinductively defined sets correspond to computable greatest fixpoint since swarm protocols are regular trees).

A swarm protocol \mathbf{G} is *invariant under event type* \mathbf{t} if either (i) \mathbf{t} does not appear in \mathbf{G} , i.e. $\mathbf{t} \notin events(\mathbf{G})$ or (ii) it only appears as part of the same choice, i.e. there is a unique $\mathbf{G}' \in subterms(\mathbf{G})$ such that $\mathbf{G}' \xrightarrow{\mathbf{c}/\mathbf{1}}$ and $\mathbf{t} \in \mathbf{1}$.

► **Definition 6.5** (Confusion-freeness). *A swarm protocol \mathbf{G} is confusion-free if \mathbf{G} is invariant for all event types in $guards(\mathbf{G})$.*

► **Example 6.6.** It is easy to check that the protocol of the running example \mathbf{G} in Example 4.2 is invariant for all types. The only type appearing in two guards is **Receipt**; however, the occurrences are associated to the same subterm. Hence, the protocol is confusion-free. ◻

6.4 Putting the pieces together

With this, we can finally state our *well-formedness* condition.

► **Definition 6.7** (Well-formedness). *A swarm protocol $\mathbf{G} = \sum_{i \in I} \mathbf{c}_i @ \mathbf{R}_i \langle \mathbf{1}_i \rangle . \mathbf{G}_i$ is well-formed with respect to a subscription σ (σ -WF for short) if*

1. \mathbf{G} is causal-consistent, determinate, and confusion-free; and
2. \mathbf{G}_i is σ -WF for all $i \in I$;

Note that well-formedness is defined coinductively and decidable on swarm protocols.

► **Example 6.8.** Examples 6.2, 6.4, and 6.6 imply that the protocol \mathbf{G} in Example 4.2 is well-formed with respect to the subscription σ given in Example 5.2. ◻

Projection preserves determinism in well-formed protocols:

► **Proposition 6.9.** *Let \mathbf{G} and σ respectively be a swarm protocol and a subscription. If \mathbf{G} is σ -WF then $\mathbf{G} \downarrow_{\mathbf{R}}^{\sigma}$ is deterministic for all \mathbf{R} .*

Well-formed swarm protocols guarantee that local machines reach eventual consensus [38] on each choice, as we show next. However, anomalies (cf. [27]) occur at system level:

- Machines could have commands enabled that would be disabled if the model were synchronous; this may lead to the emission of events that need to be ignored later.
- Events are ignored according to their type only, therefore even after full propagation of the events in the global log a machine may process events stemming from the anomalous invocation of a command.

We note that the first anomaly above is inherent to local-first architecture requirements.

The second anomaly can be avoided by a runtime system that tracks full causality information. We chose to not require full causality tracking since it imposes additional storage, communication, and computation requirements on the implementation. Our weaker causality model supports deployment on less capable hardware where needed.

7 Correct Realisations of swarm protocols

We now turn our attention to the formal characterisation of *correct implementations* of swarm protocols. As discussed in the previous sections, we deviate from the usual expected properties of mainstream (multiparty) session types, such as communication safety, session fidelity, and progress (i.e., absence of deadlocks or its variants). We first note that there are no communication mismatches in our model because every machine simply ignores unexpected or unwanted events (recall the definition of δ in Section 2.1). Session fidelity instead advocates implementations that behave as described by their types, which customarily means that the states of all components are always aligned with the global state of the protocol. Contrastingly, we aim to tolerate deviations provided that all machines eventually agree on the state of the execution of the protocol. In our setting, an implementation may be correct even if machines temporarily diverge, executing different branches of the protocol; this is quite expected if we allow independent decisions taken based on incomplete views of the global state. Consequently, correct implementations may perform sequences of commands – and hence generate logs – that are different from those derived from the corresponding protocol. In such cases, we still expect machines to be able to eventually agree on an interpretation of the log that matches one possible execution of the specification.

We tackle this problem by first defining the relevant events of an execution, namely those that are part of the *effective log*. Based on this, we establish an equivalence relation on logs that allows us to characterise the logs that can be produced by an execution of a swarm protocol's realisation as a swarm. Armed with these tools we then state that all correct realisations produce valid effective logs and that all swarm protocol executions have corresponding swarms that realise them.

7.1 Eventual fidelity

We start by introducing some machinery for making precise the notion of correct implementation of a swarm protocol.

Roughly, one may think that (\mathbf{S}, ϵ) is a faithful implementation of a swarm protocol \mathbf{G} if it produces only global logs that can be generated by \mathbf{G} . However, this notion is too strong for our setting; in fact, we appeal to a weaker notion of fidelity such that for any global log l produced by (\mathbf{S}, ϵ) , i.e. $(\mathbf{S}, \epsilon) \Longrightarrow (\mathbf{S}, l)$, there is a *related* log l' that \mathbf{G} admits, i.e. $(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l')$. We postpone for a moment the formal definition of the expected relation between logs, and convey some intuitions in the following example.

► **Example 7.1.** Consider the swarm protocol \mathbf{G} in Example 4.2, and the swarm $(\mathbf{P}, \epsilon) \mid (\mathbf{T}, \epsilon) \mid (\mathbf{0}, \epsilon) \mid (\mathbf{T}, \epsilon) \mid \epsilon$ having three taxis dubbed A , B , and C . The swarm can produce the global log

$$l_{\text{auc}} = \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{selected} \cdot \text{bid}_C \cdot \text{bidderid}_C \cdot \text{passengerid}$$

Contrastingly, \mathbf{G} cannot generate such log; in fact, the protocol continuation after generating the prefix $l_1 = \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A$ is $\delta(\mathbf{G}, l_{\text{auc}}) = \mathbf{G}_{\text{Bid}}$; hence, log l_1 can only grow by appending $\text{bid}_C \cdot \text{bidderid}_C$ or $\text{selected} \cdot \text{passengerid}$. In the second case, we obtain the log $l_2 = \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{selected} \cdot \text{passengerid}$. Remarkably, all the machines discard the events bid_C and bidderid_C when processing l_{auc} , i.e., they behave as if they were processing l_2 . In fact, $\delta(\mathbf{P}, l_{\text{auc}}) = \delta(\mathbf{P}, l_2)$, $\delta(\mathbf{T}, l_{\text{auc}}) = \delta(\mathbf{T}, l_2)$ and $\delta(\mathbf{0}, l_{\text{auc}}) = \delta(\mathbf{0}, l_2)$. \lrcorner

As highlighted by the previous example, despite the actual log generated by the swarm differing from the logs generated by the protocol, all the machines are able to consistently discard those ill-generated events after complete propagation. In other words, the states of the machines in the swarm depend only on a subset of the events in the log. We characterise such subset via a type, called *effective type*. Intuitively, the effective type of a log is the type of the sublog containing all those events that are effectively relevant to the machines in the swarm. Given a protocol G and a subscription σ , we expect an implementation to process only those events which some role has been subscribed to; consequently, our notion of *effective type* is relative to a subscription. However, the effective type of a log is not just the projection of its type with respect to the image of the subscription σ . This is illustrated in the following example.

► **Example 7.2** (Effective type and projection). The type of the log l_{auc} in Example 7.1 is

$$l_{\text{auc}} = \text{Requested} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Selected} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{PassengerID}$$

which differs from the type of l_2 which is $\text{Requested} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Bid} \cdot \text{BidderID} \cdot \text{Selected} \cdot \text{PassengerID}$. Let us consider a subscription σ such that $\sigma(P) \supseteq \{\text{Requested}, \text{Bid}, \text{BidderID}, \text{Selected}, \text{PassengerID}\}$. Then, if we just keep the sublog of l_{auc} containing all types for which at least one role has been subscribed to, then we obtain exactly l_{auc} , which does not reflect the type of the sublog that is effectively processed by the machines. For this reason, the effective type depends also on the protocol being implemented. \lrcorner

► **Definition 7.3** (Effective type). *The effective type of a log l with respect to a swarm protocol G and a subscription σ , written $\mathbb{T}_\sigma(l, G) = \mathbb{T}_\sigma(l, G, \epsilon)$, is defined as follows*

$$\mathbb{T}_\sigma(\epsilon, G, \mathbf{1}) = \epsilon \quad (2)$$

$$\mathbb{T}_\sigma(e \cdot l, G, \epsilon) = \mathbf{t} \cdot \mathbb{T}_\sigma(l, G', \mathbf{1}') \quad \text{if } \vdash e : \mathbf{t}, \mathbf{t} \in \sigma(\text{roles}(G, \sigma)), G \xrightarrow{c/\mathbf{t} \cdot \mathbf{1}} G', \text{ and } \mathbf{1}' = \text{filter}(\mathbf{1}, \sigma(\text{active}(G'))) \quad (3)$$

$$\mathbb{T}_\sigma(e \cdot l, G, \mathbf{t} \cdot \mathbf{1}) = \mathbf{t} \cdot \mathbb{T}_\sigma(l, G, \mathbf{1}) \quad \text{if } \vdash e : \mathbf{t} \quad (4)$$

$$\mathbb{T}_\sigma(e \cdot l, G, \mathbf{1}) = \mathbb{T}_\sigma(l, G, \mathbf{1}) \quad \text{otherwise} \quad (5)$$

As expected, the effective type of an empty log is the empty log type. The effective type of a non-empty log keeps track of those events that match the type of a log that can be generated by the protocol (and discards all ill-ordered events). Hence, the effective type of $e \cdot l$ with respect to G records the type \mathbf{t} of the first event e only if it has a type that is expected by the protocol (namely at least one of the roles in the protocol is subscribed to that type). According to case (3), a protocol G expects some event whose type \mathbf{t} coincides with the guard of one of its branches and at least one role is subscribed to \mathbf{t} . In such case, the effective type of the remaining log l is processed first by consuming events of type $\mathbf{1}'$ (rule (4)), which is the sequence of the remaining types generated by the branch that are observed by the *active* roles in the continuation G' , followed by considering the continuation G' . We remark here that only the types of events that are relevant for active roles are reflected in the effective type (more details are given in Section 7.2.1). Note that the types that are not observed are just disregarded from the effective type as per (5). We have that $\mathbb{T}_\sigma(l, G)$ is a well-defined function over deterministic swarm protocols because log-determinism (Definition 4.1) ensures that at most one branch of G can match the event type \mathbf{t} of the event e .

► **Example 7.4.** Let G_{auction} and G_{choose} the swarm protocol in Example 4.2. We have

$$G_{\text{auction}} \xrightarrow{\text{Offer}@\mathbf{T}(\text{Bid} \cdot \text{BidderID})} G_{\text{auction}} \quad (6) \quad G_{\text{auction}} \xrightarrow{\text{Select}@\mathbf{T}(\text{Selected} \cdot \text{PassengerID})} G_{\text{choose}} \quad (7)$$

15:20 Behavioural Types for Local-First Software

Let us compute the effective type of the log $l = \text{bid}_A \cdot \text{bidderid}_A \cdot \text{rating} \cdot l'$ on $\mathbf{G}_{\text{auction}}$ using a subscription σ for which \mathbf{P} , \mathbf{T} and \mathbf{O} are subscribed to all events but rating . We have

$$\begin{aligned} \mathbb{T}_\sigma(l, \mathbf{G}_{\text{auction}}) &= \mathbf{Bid} \cdot \mathbb{T}_\sigma(\text{bidderid}_A \cdot \text{rating} \cdot l', \mathbf{G}_{\text{auction}}, \mathbf{BidderID}) \\ &= \mathbf{Bid} \cdot \mathbf{BidderID} \cdot \mathbb{T}_\sigma(\text{rating} \cdot l', \mathbf{G}_{\text{auction}}, \epsilon) \\ &= \mathbf{Bid} \cdot \mathbf{BidderID} \cdot \mathbb{T}_\sigma(l', \mathbf{G}_{\text{auction}}, \epsilon) \end{aligned}$$

where the first equality holds by (3) since $\mathbf{Bid} \in \sigma(\mathbf{T})$ and (6), the second equality holds by (4) since $\vdash \text{bidderid}_A : \mathbf{BidderID}$, and the third equation holds by (5) since by hypothesis $\text{rating} \notin \sigma(\mathbf{P}) \cup \sigma(\mathbf{T}) \cup \sigma(\mathbf{O})$. If $l' = \epsilon$ then $\mathbb{T}_\sigma(l, \mathbf{G}_{\text{auction}}) = \mathbf{Bid} \cdot \mathbf{BidderID}$ by (2).

Suppose instead that $l' = \text{selected} \cdot \text{passengerid} \cdot \text{bid}_B \cdot \text{bidderid}_B$. Then, similarly to the first two equations above (using (7)), we have

$$\mathbb{T}_\sigma(l', \mathbf{G}_{\text{auction}}, \epsilon) = \mathbf{Selected} \cdot \mathbf{PassengerID} \cdot \mathbb{T}_\sigma(\text{bid}_B \cdot \text{bidderid}_B, \mathbf{G}_{\text{choose}}, \epsilon)$$

And, by (4), $\mathbb{T}_\sigma(\text{bid}_B \cdot \text{bidderid}_B, \mathbf{G}_{\text{choose}}, \epsilon) = \mathbb{T}_\sigma(\epsilon, \mathbf{G}_{\text{choose}}, \epsilon) = \epsilon$. \lrcorner

The relation between logs of an implementation with those of a specification that we need is the equivalence induced by the equality of their effective types.

► **Definition 7.5** (Log equivalence). *Two logs l and l' are equivalent with respect to a swarm protocol \mathbf{G} and a subscription σ , written $l \equiv_{\mathbf{G}, \sigma} l'$, if they have the same effective type with respect to \mathbf{G} and σ , i.e., $\mathbb{T}_\sigma(l, \mathbf{G}) = \mathbb{T}_\sigma(l', \mathbf{G})$.*

Then, the notion of correct implementation is simply stated as follows.

► **Definition 7.6** (Eventual fidelity). *A swarm (\mathbf{S}, ϵ) is eventually faithful to a swarm protocol \mathbf{G} and a subscription σ if $(\mathbf{S}, \epsilon) \implies (\mathbf{S}, l)$ implies $(\mathbf{G}, \epsilon) \implies (\mathbf{G}, l')$ and $l \equiv_{\mathbf{G}, \sigma} l'$ for a log l' .*

7.2 Implementation correctness by projection

Our projection operation yields an effective procedure for obtaining correct implementations out of well-formed swarm protocols which we call *realisations*.

► **Definition 7.7** (Realisation). *Let \mathbf{G} be a swarm protocol and σ be a subscription. A realisation (of size n) of \mathbf{G} with respect to σ , shortened as (σ, \mathbf{G}) -realisation, is a swarm (\mathbf{S}, ϵ) of size n such that, for each $1 \leq i \leq n$, there exists a role $\mathbf{R} \in \text{roles}(\mathbf{G}, \sigma)$ such that $\mathbf{S}(i) = (\mathbf{G} \downarrow_{\mathbf{R}}^\sigma, \epsilon)$. A realisation \mathbf{S} is complete if for all $\mathbf{R} \in \text{roles}(\mathbf{G}, \sigma)$ there exists $1 \leq i \leq n$ such that $\mathbf{S}(i) = (\mathbf{G} \downarrow_{\mathbf{R}}^\sigma, \epsilon)$; we call partial a realisation which is not complete.*

Remarkably, the number of machines in a realisation is not related to the number of roles in the corresponding swarm protocol. Indeed, Definition 7.7 simply requires that each machine in the swarm plays one of the roles in the swarm protocol. Concretely, we may have several components implementing the same role (i.e., the role is replicated) as well as roles without a corresponding machine, that is partial realisations.

► **Example 7.8** (Realisations). The swarm protocol in Example 4.2 would typically be realised by one machine for the passenger $\mathbf{G} \downarrow_{\mathbf{P}}^\sigma$, several taxis running the machine $\mathbf{G} \downarrow_{\mathbf{T}}^\sigma$, and at least one accounting office running $\mathbf{G} \downarrow_{\mathbf{O}}^\sigma$. A partial realisation could be one without an accounting office, in which case no machine can generate **receipt** events. \lrcorner

The rest of this section is devoted to showing that realisations (either complete or partial) are eventually faithful if they are obtained by projecting well-formed swarm protocols.

7.2.1 Projections and effective types

We first establish a correspondence between the behaviour of a single projection and that of the respective protocol. In particular, we show that effective types provide an accurate abstraction of the information contained in a log that is relevant for a role. Concretely, the next result states that a projection enables a command after processing a log l only when the protocol enables the same command after producing an equivalent log l' .

► **Lemma 7.9.** *If G is a σ -WF swarm protocol and $(\delta(G \downarrow_R^\sigma, l)) \downarrow_{c/1}$ then there exists $l' \equiv_{G,\sigma} l$ such that $(G, \epsilon) \Longrightarrow (G, l')$ and $\delta(G, l') \xrightarrow{c/1} G'$.*

Apparently equivalent logs are indistinguishable for a machine, i.e., $l \equiv_{G,\sigma} l'$ implies $\delta(G \downarrow_R^\sigma, l) = \delta(G \downarrow_R^\sigma, l')$. However this might not be the case if logs do not include all the events generated by the same command as shown in the next example.

► **Example 7.10.** Consider the swarm protocol $G = c@R\langle a \cdot b \rangle . d@S\langle c \rangle . 0$ with $\sigma = \{R \mapsto \{a, b\}, S \mapsto \{a, c\}\}$. If $\vdash a : a$ and $\vdash b : b$ then $\mathbb{T}_\sigma(a, G) = a$ and $\mathbb{T}_\sigma(a \cdot b, G) = a$; in fact the first equation holds by definition and the second holds because $b \notin \sigma(\text{active}(d@S\langle c \rangle . 0')) = \sigma(S) = \{a, c\}$. Therefore, $a \equiv_{G,\sigma} a \cdot b$. Now take the projection of G over R with respect to σ , i.e., $M_R = G \downarrow_R^\sigma = a?b?(d@S\langle c \rangle . 0 \downarrow_R^\sigma) = a?b?0$. Clearly $\delta(M_R, a \cdot b) \neq \delta(M_R, a)$. ◻

Two considerations on Example 7.10 are worthwhile. On the one hand, while $a \cdot b$ has all the events produced by the execution of the command c , the log consisting of just the event a does not. Since we assume that all events are eventually propagated, our technical development in the next section will disregard incomplete (global) logs. On the other hand, one may wonder about the fact that effective types do not collect information of events that are not observed by active roles. This is essential to account for the fact that a realisation may interject events. For instance, a realisation may actually generate a log of type $a \cdot c \cdot b$ because a machine that implements the role S may perform the command d as soon as it processes an event of type a ; hence the generated event can precede the one of type b in the consolidated log. If our notion of log equivalence were fine enough to distinguish logs of type $a \cdot c \cdot b$ from $a \cdot b \cdot c$ then we would rule out implementations behaving as above, which is not what we want because the interaction does not violate the protocol.

7.2.2 Characterisation of the logs admitted by a protocol

We now provide a characterisation of the logs that can be generated by a realisation of a well-formed swarm protocol. To do this we have to take into account for the possible reordering and the spurious events that can be generated by machines that faithfully implement a protocol. Intuitively, we may think that a realisation generates logs that correspond to the combination of several executions of the protocol, which might share a common prefix. Consider again the swarm protocol G in Example 4.2. As discussed in Example 7.1, we expect realisations to be able to generate logs such as l_{auc} of this example. Note that such log can be generated by merging, among others, two different reductions of G , e.g. $(G, \epsilon) \Longrightarrow (G, \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_A \cdot \text{bidderid}_A \cdot \text{selected} \cdot \text{passengerid})$ and $(G, \epsilon) \Longrightarrow (G, \text{requested} \cdot \text{bid}_B \cdot \text{bidderid}_B \cdot \text{bid}_C \cdot \text{bidderid}_C)$. Note that the reductions share events (accounting for an scenario in which the computation has diverged). Intuitively, two runs can be combined either if they produce disjoint logs or they share events that come from a common execution prefix (as in the previous example). Formally, two runs $(G, \epsilon) \Longrightarrow (G, l_i)$ with $i = 1, 2$ are *consistent* if there exist logs $l, l'_1 \cap l'_2 = \emptyset$, such that $l_i = l \cdot l'_i$ and $(G, \epsilon) \Longrightarrow (G, l) \Longrightarrow (G, l \cdot l'_i)$

for $i = 1, 2$. The notion of consistency is lifted to sets of runs $\{(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l_i)\}_{1 \leq i \leq k}$, by requiring pair-wise consistency. We write l_i^j for the sequence of events produced by the j -th step in the reduction i , i.e., $(\mathbf{G}, \epsilon) \Longrightarrow^{j-1} (\mathbf{G}, l_i^j) \xrightarrow{c/1} (\mathbf{G}, l_i^j \cdot l_i^j) \Longrightarrow (\mathbf{G}, l_i)$.

► **Definition 7.11** (Admissible log). *A log l is admissible for a σ -WF protocol \mathbf{G} if there are consistent runs $\{(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l_i)\}_{1 \leq i \leq k}$ and a log $l' \in (\bowtie_{1 \leq i \leq k} l_i)$ with $l' \equiv_{\mathbf{G}, \sigma} l = \bigcup_{1 \leq i \leq k} l_i$, and for all $1 \leq i \leq k$ and $l_i^j \sqsubseteq l$ for all events l_i^j produced by the j -th step in reduction i .*

Remarkably, admissible logs are not just those that can be obtained by merging several logs l_i ; it may be the case that l is admissible but $l \notin (\bowtie_{1 \leq i \leq k} l_i)$. In fact the notion is weaker and accounts for the possible reorderings of events that do not change the effective type of the log. Consider the protocol in Example 7.10 and its complete realisation consisting of two machines. As previously discussed, that realisation may generate a log of type $a \cdot c \cdot b$. With a single run of the protocol, i.e. by fixing $k = 1$ and taking $(\mathbf{G}, \epsilon) \xrightarrow{c/a \cdot b} (\mathbf{G}, a \cdot b) \xrightarrow{d/c} (\mathbf{G}, l_1)$ with $l_1 = a \cdot b \cdot c$, we can conclude that $a \cdot c \cdot b$ is generated by some realisation. Note that $\bowtie_{1 \leq i \leq 1} l_i = \{l_1\} = \{a \cdot b \cdot c\}$. Hence, $l' \in \bowtie_{1 \leq i \leq 1} l_i$ iff $l' = a \cdot b \cdot c$ and $\mathbb{T}_\sigma(l', \mathbf{G}) = a \cdot c$. Then, the log $a \cdot c \cdot b$ is equivalent (i.e., it has the same effective type), and moreover it has the same elements and preserves the relative order between events generated by the same command (i.e., a precedes b). Hence, we conclude that the protocol admits the log $a \cdot c \cdot b$. On the contrary, the last condition $l_i^j \sqsubseteq l$ about the preservation of the relative order of events generated by the same command bans logs such as $b \cdot a \cdot c$.

Next lemma ensures that any admissible log of a well-formed protocol is equivalent to a log obtained by the sequential execution of the protocol. Namely, despite a log may contain events produced by conflicting decisions, its effective type corresponds to a sequential run.

► **Lemma 7.12.** *If l is admissible for a σ -WF swarm protocol \mathbf{G} then there exists a log l' such that $(\mathbf{G}, \epsilon) \Longrightarrow (\mathbf{G}, l')$ and $l \equiv_{\mathbf{G}, \sigma} l'$.*

Moreover, if we extend an admissible log with events generated by the execution of command enabled over a partial view of the global log, then we obtain an admissible log. This property is instrumental for our main result in the following section (Theorem 7.14).

► **Lemma 7.13.** *Let l_1 and $l_2 \subseteq l_1$ be admissible logs for a σ -WF swarm protocol \mathbf{G} . If $(\mathbf{G}, l_2) \xrightarrow{c/1} (\mathbf{G}, l_2 \cdot l_3)$ and $l \in l_1 \bowtie (l_2 \cdot l_3)$ then l is admissible for \mathbf{G} .*

7.2.3 Realisations are faithful

Our key result shows that realisations of well-formed protocols only generate admissible logs.

► **Theorem 7.14.** *Let (\mathbf{S}, ϵ) be a realisation of a σ -WF swarm protocol \mathbf{G} . If $(\mathbf{S}, \epsilon) \Longrightarrow (\mathbf{S}', l)$ then l is admissible for \mathbf{G} .*

Since every admissible log is equivalent to a log generated by the protocol (Lemma 7.12), we conclude that any realisation of a well-formed swarm protocol is eventually faithful (i.e., correct). Note that this implies that all realisations of a well-formed swarm protocol exhibit eventual consensus [38] regarding which branch is taken in its choices (concretely, there is a $\xrightarrow{\tau}$ step after which all machines take the same branch in their state computation).

► **Corollary 7.15.** *Every realisation of size n of a σ -WF swarm protocol \mathbf{G} is eventually faithful with respect to \mathbf{G} and σ .*

The above result is independent from the number of replicas that implement each role; it also holds for partial realisations (i.e., when some roles are absent).

7.3 Implementation completeness

Differently from common session type systems, the behaviour of a complete realisation (i.e., one in which every role is implemented) is complete with respect to the protocol, in the sense that every reduction of the protocol can be mimicked by the realisation. This derives from the fact that non-determinism in our model arises from the execution of external commands but not because of the abstraction of internal (and customary deterministic) choices. Firstly, we note that logs that are generated sequentially according to the protocol drives the machines to the corresponding states.

► **Lemma 7.16.** *If G is σ -WF swarm protocol and $(G, \epsilon) \Longrightarrow (G, l)$ then $\delta(G \downarrow_R^\sigma, l) = \delta(G, l) \downarrow_R^\sigma$ for all $R \in \text{roles}(G, \sigma)$.*

Moreover, Proposition 7.17 below states that a complete realisation is able to generate the logs that are generated by the protocol (the result is obtained by using previous result and by propagating all events to all replicas right after a machine performs a command).

► **Proposition 7.17.** *Let (S, ϵ) be a complete realisation of size n of the σ -WF swarm protocol G . If $(G, \epsilon) \Longrightarrow (G, l)$ then there is a swarm S' such that $(S, \epsilon) \Longrightarrow (S', l)$.*

8 Related work

It is widely accepted that solutions to distributed coordination problems strongly depend on the adopted computation model [15, 21, 2]. Our proposal is grounded on the principles of *local-first cooperation* [26, 25]. Key to this architecture is the *autonomy* of each participating node within a swarm. Autonomy allows each node to make progress independently of network connections, availability of other nodes, or delay in the communications. Our target model features specific hues that distinguish it from other behavioural types systems. In our case, distributed heterogeneous components interact asynchronously by emitting and consuming *events* according to a role specified in a given protocol (such as passenger and the taxi in our running example). More precisely, events are the side effects of commands non-deterministically executed by components; events are locally logged by each of the components and asynchronously spread through the swarm. Crucially, we do not make any assumption on the relative speed of communications and simply require that logs *eventually* agree on the order of events [4]. This liberal setting permits inconsistencies: components may take discordant decisions which compromise the execution of the protocol and exhibit behaviour precluded in strongly consistent models. Our approach lies within methods related to data replication, which are notoriously complex. In fact, standard techniques have to trade-off among availability, consistency, and partition tolerance [16]. Several techniques have been proposed, such as conflict-free replicated data types [34], cloud types [5], consistency contracts [35], invariants [18, 24, 3], linearizability [40], and operational models for applications such as GSP [6, 17]. An original facet of our approach is that we use behavioural types to discipline data replication in order to eventually reach consistency. We focus on the consequences that arise from the ability of each node to take decisions based exclusively on *local information*.

Our proposal is inspired by the choreographic framework introduced in the seminal work on *multiparty session types* [19, 20]. However, the peculiarities of our execution model as well as on the properties that we target require a radical change in the definition of well-established notions of global types, such as projections and well-formedness. The main originality of our approach is that components speculatively proceed along several (possibly inconsistent)

branches of distributed choices provided that an agreement is eventually reached. Intuitively, this is attained by disregarding all executions bar one when the local logs “consolidate”, namely when relevant events have propagated to all relevant components. As far as we are aware of, multiple selectors are forbidden in the well-formedness conditions of most behavioural type systems [21]. A slight weakening of this condition is given in [23] but the conditions there still reject the protocol in Example 1.1. Noteworthy, we divert from the research path of behavioral types with respect to the properties we are after. We aim to guarantee that projections of global specifications yield realisations of swarms that eventually reach a consistent view of the distributed execution, even in presence of transitory inconsistencies. This is in contrast with behavioural type systems designed to attain (dead)lock freedom or some notions of progress (see [21] and [10] for a recent account on the binary case).

Secondly, our behavioural specifications completely abstract over the number of instances enacting a role. This is often not the case for multiparty session types. Parametric multiparty session types have been considered in [41, 11, 12] and more recently in [8, 23]. These proposals aim to capture the fact that roles in a protocol are “connected” to form a topology that can be generalised (e.g. parameterising a ring topology by its size). These behavioural type systems therefore require to explicitly handle the parameters of the protocol. Our specifications are instead completely oblivious of such parameters. To the best of our knowledge, multiparty session types have focused on point-to-point, message-passing communication model, even to deal with highly dynamic scenarios, as those involving robot coordination [29].

9 Final Remarks

We proposed rather unconventional behavioural types deviating from point-to-point, message-passing communication, which is a common practice (see e.g. [13, 39, 21]). Components in our setting interact via a shared distributed log built without any further coordination mechanism. More precisely, each component keeps a local, possibly partial and inconsistent view of the global log. Based on that view alone a component may perform an action with immediate effect on its local state; those effects are then propagated asynchronously to the rest of the system. This implies that components can perform globally invalid actions (as long as they are locally valid), but we require every component be able to recognise these and eventually behave as if only valid actions were performed. Technically this means that we renounce established properties like *session fidelity* to guarantee instead that systems eventually agree without dedicated coordination (our typing discipline guarantees deadlock-freedom, though).

Our target applications intrinsically involve sets of components whose number is statically unknown: components may dynamically join and leave the execution of an interaction. A reference application domain is factory logistics where all the assumptions listed at the end of Section 1 apply: collaborative components act in a trustworthy setting, compensations are specified for irreversible actions, and the underlying communication infrastructure is controlled by the business owner. The collaborative assumption, while commonplace in the literature on behavioural types, may be unrealistic in some domains. Extension to adversarial settings will for example require enforcing that machines cannot violate causality when emitting events, i.e. they cannot artificially truncate their local log to undo an earlier choice via an event that they maliciously sort before it. This can be achieved by requiring them to cryptographically sign their logs, allowing other nodes to prove illicit behaviour; similarly, machines joining later would be required to sign recent events before being allowed to emit.

Common practice in behavioural types is to describe protocols in terms of the roles that components may play. Unlike most behavioural types, ours are agnostic to the number of instances playing each role. We assume that any role in any swarm protocol can be replicated

as many times as needed. This choice also impacts the interpretation of choices. In standard approaches, every choice is assigned to a single role implemented by a single component responsible for coordinating the decision. This is problematic when the implementation of a role can be replicated, even more when the states of the replicas may be misaligned: different components may decide differently. Consequently, choices in our swarm protocols are intended to be resolved distributedly among components that may implement different roles. Our solution is based on speculative computation: different choices can proceed concurrently until components are able to agree (by inspecting their local state) on the branch that has been selected based on a total order for all events (implemented for example using Lamport timestamps and unique node identifiers).

Our computational model gives an abstract description of Actyx infrastructure [1], in which machines are actually implemented as programs in `TypeScript`: the machines presented here play the role of local types that describe the intended behaviour of each component.

Subscriptions can be seen as a minimum requirement for which events need to be available to each participant in a swarm protocol. This directly translates to which events need to (or shall) be sent to a swarm member participating in a given protocol session. Swarm members not partaking in the session do not need to see any of the events (e.g. other passengers), so the middleware should not send them there (this can be elevated to a security guarantee if needed). Our system does not cause additional information to be sent, it is minimal within the constraints of our well-formedness conditions (which are sufficient but not necessary, so there is some room for further improvement).

Determining a suitable subscription could be hard in general. We conjecture that the swarm protocol itself could be used to infer a suitable “minimal” registration enforcing well-formedness to be suggested to designers. Such subscription could then manually be refined, provided that well-formedness is preserved. Moreover, we may envision programmers specifying just the relevant information that needs to be transmitted and then automatically infer the events needed for coordinating choices (pretty much in the style of the communication of labels in session types).

An underlying assumption about speculative execution is that the effects of performing invalid actions can be discarded. In other words, invalid actions have no consequences. In several situations this may be unacceptable. Swarm protocols could be used to systematically identify such situations – e.g. by noting when corresponding events are disregarded by machines – and enable principled treatment at the application level. We plan to study suitable extensions for our projections that automatically inject the required behaviour for executing amending actions. Alternatively, we will explore monitoring approaches equipped with sanitisers responsible for compensation.

We have only partly addressed failures in our model: while we do model transient inability to receive (which would inhibit the event propagation transition [PROP]) or to operate (i.e. inhibit command invocation [LOCAL]), we do not model permanent inability to send. In the presence of a stop failure a machine could communicate the first event of a choice but then fail in propagating all the expected following events resulting from the command invocation, in which case the system could get stuck. A fix for this issue could be to only proceed with an external choice once all specified events are present in the local log, allowing the swarm to permanently discard a choice made by a failing machine; this could be expressed by ingesting logs instead of single events in the definition of machine semantics.

Another worthwhile extension, hinted at in Section 2.3, is to achieve a per-choice notion of non-interference if the first event of a choice not only decided which branch to take but also from which source machine to consume the rest of the choice’s events. This would

further strengthen the failure handling sketched above by making sure that inputs from failing machines are consistently discarded. Characterising the precise guarantees that derive from such a scheme will be an interesting topic for further study.

References

- 1 Actyx AG. Actyx developer website, 2020-2022. accessed 2022/07/06. URL: <https://developer.actyx.com>.
- 2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.
- 3 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno M. Preguiça. IPA: invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, 2018. doi:10.14778/3297753.3297760.
- 4 Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1–2):1–150, October 2014. doi:10.1561/2500000011.
- 5 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 283–307, Berlin, Heidelberg, 2012. Springer.
- 6 Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 568–590, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2015.568.
- 7 Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(5):1–61, 2009.
- 8 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019. doi:10.1145/3290342.
- 9 Bruno Courcelle. Fundamental properties of infinite trees. *Theor. Comput. Sci.*, 25:95–169, 1983. doi:10.1016/0304-3975(83)90059-2.
- 10 Ornela Dardha and Jorge A. Pérez. Comparing type systems for deadlock freedom. *J. Log. Algebraic Methods Program.*, 124:100717, 2022. doi:10.1016/j.jlamp.2021.100717.
- 11 Pierre-Malo Deniérou and Nobuko Yoshida. Dynamic multirole session types. In Thomas Ball and Mooly Sagiv, editors, *POPL*, pages 435–446. ACM, 2011. doi:10.1145/1926385.1926435.
- 12 Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.
- 13 Mariangiola Dezani-Ciancaglini and Ugo de’Liguoro. Sessions and session types: An overview. In *WS-FM*, volume 6194 of *LNCS*, pages 1–28. Springer, 2009. doi:10.1007/978-3-642-14458-5_1.
- 14 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.
- 15 Simon Gay and Antonio Ravara, editors. *Behavioural Types: from Theory to Tools*. Automation, Control and Robotics. River, 2009.
- 16 Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

- 17 Alexey Gotsman and Sebastian Burckhardt. Consistency Models with Global Operation Sequencing and their Composition. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2017.23.
- 18 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. ‘Cause i’m strong enough: reasoning about consistency choices in distributed systems. In *POPL 2016*, pages 371–384, 2016.
- 19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, pages 273–284, 2008.
- 20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM (JACM)*, 63(1):1–67, 2016.
- 21 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.
- 22 David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.
- 23 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In Peter Müller, editor, *Programming Languages and Systems*, pages 251–279, Cham, 2020. Springer International Publishing.
- 24 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, 2018.
- 25 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 154–178, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.
- 26 Roland Kuhn. Local-first cooperation: Autonomy at the edge, secured by crypto, 100% available, 2021. accessed 2021/06/20. URL: <https://www.infoq.com/articles/local-first-cooperation/>.
- 27 Roland Kuhn, Hernán Melgratti, and Emilio Tuosto. Behavioural types for local-first software, 2023. arXiv:2305.04848.
- 28 Roland Kuhn, Hernán Melgratti, and Emilio Tuosto. Behavioural Types for Local-First Software (Artifact). *Dagstuhl Artifacts Series*, 9(2):14:1–14:5, 2023. doi:10.4230/DARTS.9.2.14.
- 29 Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. Motion session types for robotic interactions (brave new idea paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 30 Microsoft. Typescript: Javascript with syntax for types, 2012-2023. accessed 2023/02/02. URL: <https://www.typescriptlang.org/>.
- 31 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- 32 Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- 33 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.
- 34 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS 2011*, pages 386–400, 2011. doi:10.1007/978-3-642-24550-3_29.

- 35 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI 2015*, pages 413–424. ACM, 2015.
- 36 Ecma TC39. Ecma-404, 2017. accessed 2023/02/13, alternative RFC8259. URL: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>.
- 37 Ecma TC39. Decorators, 2022. accessed 2023/02/02. URL: <https://github.com/tc39/proposal-decorators>.
- 38 Lewis Tseng. Eventual consensus: Applications to storage and blockchain : (extended abstract). In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 840–846, 2019. doi:10.1109/ALLERTON.2019.8919675.
- 39 Vasco T. Vasconcelos. Fundamentals of session types. *Inf. and Comp.*, 217:52–70, 2012. doi:10.1016/j.ic.2012.05.002.
- 40 Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. Replication-aware linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 980–993, 2019.
- 41 Nobuko Yoshida, Pierre-Malo Deniérou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010. doi:10.1007/978-3-642-12032-9_10.

Constraint Based Compiler Optimization for Energy Harvesting Applications

Yannan Li¹ ✉

University of Southern California, Los Angeles, CA, USA

Chao Wang ✉

University of Southern California, Los Angeles, CA, USA

Abstract

We propose a method for optimizing the energy efficiency of software code running on small computing devices in the Internet of Things (IoT) that are powered exclusively by electricity harvested from ambient energy in the environment. Due to the weak and unstable nature of the energy source, it is challenging for developers to manually optimize the software code to deal with mismatch between the intermittent power supply and the computation demand. Our method overcomes the challenge by using a combination of three techniques. First, we use static program analysis to automatically identify opportunities for *precomputation*, i.e., computation that may be performed ahead of time as opposed to just in time. Second, we optimize the precomputation policy, i.e., a way to split and reorder steps of a computation task in the original software to match the intermittent power supply while satisfying a variety of system requirements; this is accomplished by formulating energy optimization as a constraint satisfiability problem and then solving the problem using an off-the-shelf SMT solver. Third, we use a state-of-the-art compiler platform (LLVM) to automate the program transformation to ensure that the optimized software code is correct by construction. We have evaluated our method on a large number of benchmark programs, which are C programs implementing secure communication protocols that are popular for energy-harvesting IoT devices. Our experimental results show that the method is efficient in optimizing all benchmark programs. Furthermore, the optimized programs significantly outperform the original programs in terms of energy efficiency and latency, and the overall improvement ranges from 2.3X to 36.7X.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Constraint and logic programming

Keywords and phrases Compiler, energy optimization, constraint solving, cryptography, IoT

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.16

Supplementary Material *Software:* <https://github.com/YannanLiCS/CouponMaker>

Funding This work was partially funded by the U.S. NSF grants CNS-1702824 and CCF-2220345.

1 Introduction

Energy harvesting is an environment-friendly technology that converts ambient energy in the environment such as sunlight, RF emission, and vibration into electricity [41, 38, 7, 33, 35, 32, 48]. When being used to power small computing devices in the Internet of Things (IoT), it avoids a main problem in the deployment of IoT at scale, which is the need to frequently change batteries [12]. Due to this reason, energy harvesting has been increasingly used in real-world deployment of IoT devices [44, 26]. However, due to the weak and unstable nature of the energy source, it is often challenging for developers to manually optimize the software code running on these IoT devices, to deal with problems caused by *mismatch* between the intermittent power supply and the often unpredictable computation demand.

¹ Corresponding author



© Yannan Li and Chao Wang;

licensed under Creative Commons License CC-BY 4.0

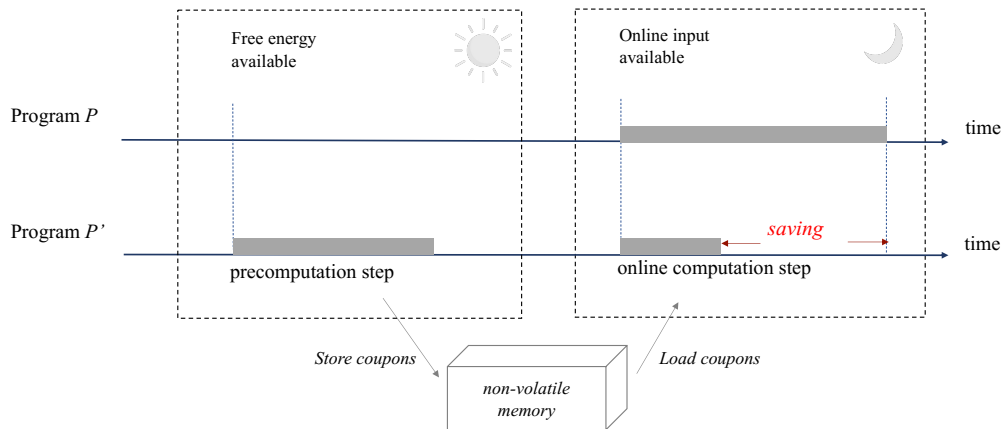
37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 16; pp. 16:1–16:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

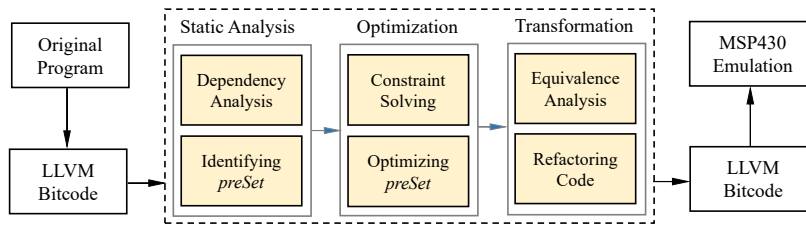


■ **Figure 1** Using precomputation to reduce the energy needed by the (online) computation task.

Consider an IoT device powered by electricity harvested from sunlight as an example. During the day time, there may be significantly more harvested electricity than the combined total of what can be stored in the supercapacitor of the device, and what can be consumed by the software code running on the device. During the night time, however, the electricity stored in the supercapacitor may be significantly less than what is needed by the software code running on the device. In this context, an important research question is whether the *mismatch between supply and demand* can be avoided, or at the very least mitigated by rewriting the original software in such a way that, while the functionality of the software remains the same, the overall energy efficiency is improved. Prior work [4, 5, 47] has demonstrated the feasibility of this approach, based on two observations made for typical energy-harvesting IoT devices.

The first observation is that, since the software on an IoT device only runs from time to time, rather than continuously, the device may be idle when ambient energy in the environment is abundant (e.g., sunlight during the day time) and yet the supercapacitor used to store the harvested electricity is full. In such a case, the *freely available* energy in the environment cannot be utilized. The second observation is that, in such an IoT device, the most common computation tasks are collecting sensor data from time to time, and encrypting these sensor data before sending them to some remote servers, e.g., servers in the cloud. Thus, the most time-consuming and energy-consuming part of the computation is the execution of the secure communication protocol. While the sensor data may have to be collected *just in time*, a significant part of the secure communication protocol (e.g., computing security tokens needed for encrypting the sensor data) may be executed *ahead of time*. This leads to the idea of leveraging the *precomputation* opportunities to utilize the *freely-available* ambient energy in the environment.

Figure 1 illustrates how to optimize a computation task that must be executed during the night time, when ambient energy is not available. While the original program (P) has to execute the entire computation task during the night time using electricity stored in the supercapacitor, the optimized program (P') executes a significant part of the task during the day time, by harvesting the freely-available ambient energy that otherwise would have to be wasted due to the storage limit of the supercapacitor. In some sense, the precomputation performed during the day time transforms the solar energy to a digital form, called *coupons*, and stores them in the non-volatile memory of the IoT device. During the night time, these coupons are used to lower the energy cost of the remaining part of the computation task.



■ **Figure 2** The overall flow of our constraint based method for energy optimization.

There are two main benefits. The first one is reduction in latency for the online computation part, since a significant portion of the computation task has been completed ahead of time. The second one is increase in the number of computation tasks that can be completed by the device. As a concrete example, Suslowicz et al. [47] show that, for a popular secure communication protocol based on *one-time pad (OTP)* [45], using precomputed OTPs for sensor data encryption reduces the energy cost of the online computation to 5% of the original energy cost needed for AES-OFB. Since the energy used to precompute OTPs is free, the overall energy reduction is close to 18 times (18X). To understand what this means, consider an IoT device that must complete 20 tasks during the day time and 20 tasks during the night time, but the electricity stored in the supercapacitor is only enough to support the completion of 2 tasks during the night time. Without precomputation, the device may be able to complete 20 tasks during the day time and only 2 tasks during the night time. By leveraging the coupons precomputed during the day time, the same device is able to complete 20 tasks and 20 partial tasks during the day time and finish off these 20 partial tasks during the night time.

However, to obtain the aforementioned benefits of precomputation, the current state of the art [4, 5, 47] requires a domain expert to optimize the software code manually, which is not only labor intensive but also error prone. Furthermore, the domain expert must be familiar with both the functionality of the software code and the energy characteristics of the hardware platform. The domain expert must also consider all of the system requirements while making the trade-off between energy reduction and increase in storage cost. In addition, manual optimization does not respond well to frequent software updates in practice: if the original software code is updated due to a bug fix or a security patch, there will be no easy way to update the manually-optimized software code.

To solve these problems, we propose a fully automated method for optimizing the energy efficiency of software running on energy-harvesting IoT devices. Toward this end, we must overcome three technical challenges. The first challenge is to identify the precomputation opportunities from the original software code automatically. The second challenge is to optimize the precomputation policy by exploiting the *energy-storage* trade-off and deciding which part of the computation task should be precomputed and which part of the computation task should be computed just in time. The third challenge is to automatically transform the software code to implement the energy optimization policy.

Figure 2 shows the overall flow of our method, which builds upon the state-of-the-art LLVM compiler platform [29]. Given the original program, our method takes three steps to produce the optimized program. In the first step, our method conducts a static analysis of the original program to identify precomputation opportunities, which are captured by *preSet* – the set of instructions in the program that may be computed ahead of time. In the second step, our method computes an optimal subset of *preSet* based on a variety of system

requirements, to minimize the energy cost while satisfying all requirements, including the storage limit of non-volatile memory used to save precomputation results. In the third step, our method leverages the LLVM compiler to generate the optimized program that has the ability to load the precomputation results from non-volatile memory and leverage them to speed up the just-in-time (online) computation part of the task. Finally, we evaluate the performance of the optimized programs on a popular hardware platform (MSP430 [24]) for energy-harvesting applications.

At the center of our method is a constraint based technique for optimizing the precomputation policy. The policy captures a solution to the complex optimization problem. The optimization problem is complex for several reasons. First, just because an instruction may be precomputed does not mean it is beneficial to precompute it, since precomputing does not always reduce energy cost; there is a trade-off between the cost of storing a precomputed coupon and the benefit of avoiding computing it directly. Second, decisions on *which instructions to precompute* cannot be made in isolation, since many of these instructions are dependent on each other; the precomputation policy has to consider all of the intra- and inter-procedural control- and data-flow dependencies in the program. Third, the size of the non-volatile memory used to store the precomputed coupons may not grow monotonically with the number of precomputed instructions, and furthermore, not all intermediate computation results in the program need to be stored as coupons in non-volatile memory. We will use concrete examples in Section 2 to illustrate these challenges and our proposed solution to overcome these challenges.

To demonstrate the effectiveness of our method, we have implemented and evaluated it on a large number of benchmark programs. Our implementation builds upon the LLVM compiler [29] and the Z3 SMT solver [11]. Specifically, we use LLVM to parse the original software code (written in the C language), conduct static program analysis, and generate the optimized software code; we use Z3 to solve the constraint satisfiability problems formulated by our method. Our tool was evaluated on 26 benchmark programs, which are C programs implementing popular secure communication protocols for IoT devices; in total, they have 31,113 lines of C code (LoC). The LoC of each program ranges from 339 to 1,572. Our target hardware platform is MSP430 [24], a family of ultra-low-power microcontroller units (MCUs) popular for energy-harvesting IoT applications.

Our experimental results are promising. In terms of the efficiency of our method, the experimental evaluation shows that all of the benchmark programs can be optimized by our tool quickly, and the optimization time is always limited to a few seconds. In terms of the effectiveness of our method, the experimental evaluation shows that all of the optimized programs significantly outperform the original programs in terms of energy efficiency and latency. Specifically, reduction in the overall energy cost ranges from 2.3X to 36.7X.

To summarize, this paper makes the following contributions:

- We propose a compiler based technique for automatically identifying precomputation opportunities in the software code using static analysis and then exploiting these opportunities using a semantic-preserving program transformation.
- We formulate energy optimization as a constraint satisfiability problem and solve the problem using an off-the-shelf SMT solver; this approach is not only flexible but also efficient in minimizing the energy cost while satisfying a variety of system requirements.
- We implement the method using a state-of-the-art compiler (LLVM) and a popular hardware platform (MSP430) for energy-harvesting applications, and demonstrate the effectiveness on a large number of benchmark programs.

2 Background

We review the technical background, including the characteristics of the hardware platform (MSP430) and an example software program to motivate our approach.

2.1 The Hardware Platform

MSP430 is a family of microcontroller units (MCUs) based on a 16-bit RISC instruction set architecture. Due to our focus on energy-harvesting applications, we are concerned with a subset of MSP430 MCUs that have the main memory partitioned into the volatile part and the non-volatile part. Depending on the application, data may be stored either in volatile memory or in non-volatile memory. These MCUs have a large number of configuration parameters, including sixteen nominal frequencies in the range 0.06 MHz to 16 MHz. For example, they may run in a low-power mode at the clock frequency of 1 MHz and the supply voltage of 1.8V, or in a high-performance mode at the clock frequency of 16 MHz and the supply voltage of 2.9V.

Since MSP430 MCUs are designed for low-power applications, they have no instruction cache or data cache. Unlike high-end CPUs widely used in servers and desktops, which routinely use advanced frequency or voltage scaling techniques, low-power MCUs such as MSP430 have significantly simpler energy models: fluctuations in power consumption are primarily due to the dynamics in supply voltage and clock speed. In fact, power consumption may be modeled using a non-linear function derived by empirically measuring the impact of varying voltage supplies and clock speeds on the power consumption of real hardware for all possible MCU configurations [2].

Accurate compile-time analysis for energy prediction [10, 3] is well studied topic for transiently powered computing systems [2], where software developers need to know the worst-case energy cost of a computation task, to maximize the software's utilization of the electricity harvested from the environment and to ensure timely checkpointing of the program state before loss of power. The accuracy of such compile-time analysis techniques have come close to direct hardware emulation. While direct hardware emulation [20, 8] offers the highest possible accuracy due to the direct measurement on target hardware, it does not offer the level of convenience and automation desired at the early stages of software development.

In this work, we evaluate our proposed method using MSPSim [15, 39], which is a popular compile-time analysis tool for MSP430. Specifically, we use MSPSim to compute and then compare the latency and energy cost of all benchmark programs, before and after our constraint-based optimization. MSPSim allows the developer to tag a piece of the software code for which energy consumption will be estimated. It does this by first generating the assembly code for MSP430, and then analyzing the assembly code to compute the number of MCU cycles needed to execute each basic block. Then, it estimates the energy consumption of each basic block based on the empirically derived energy model, the supply voltage, and the clock speed of the device.

At a high level, the energy consumption depends on the supply voltage as well as the electrical current for a given resistance of the MCU, the latter of which in turn depends on the supply voltage and the clock speed. For more details on the energy model used in such compile-time analysis tools, refer to Ahmed et al. [2].

```

1  __interrupt void ISR(void) {
2      if (msg_ready) {
3          wots(msg, pub_key, sig);
4          //Send the pair <pub_key,sig> to verifier;
5      }
6  }
7  void wots(MSG msg, KEY pub_key, SIG sig) {
8      gen_key(priv_key, pub_key);
9      sign(msg, priv_key, sig);
10 }

```

■ **Figure 3** An example program that invokes the W-OTS routine when `msg` is ready. Here, `msg_ready` and `msg` are global variables updated by other functions not presented in this figure. For `wots()`, `msg` is the input while `pub_key` and `sig` are the output. For `gen_key()`, both `priv_key` and `pub_key` are the output. For `sign()`, `msg` and `priv_key` are the input while `sig` is the output.

2.2 The Software Program

Figure 3 shows the program, where `ISR` stands for the interrupt service routine. Assume that the routine is triggered periodically by a timer. Whenever the input data stored in `msg` is ready, the subroutine `wots()` is invoked (Line 3). It implements a hash-based cryptographic primitive called the *Winternitz* one-time signature (W-OTS [40]). Here, `msg` is the input, while `pub_key` and `sig` are the output. After generating the output, the device sends the pair (`pub_key,sig`) to a verifier on a remote server (Line 4).

Let us take a closer look at the routine `wots()` defined in Lines 7-10, which consists of two subroutines. The subroutine `gen_key()` is invoked first, which returns a fresh pair of the private key `priv_key` and the public key `pub_key` as output. Then, the subroutine `sign()` is invoked, which takes `msg` and `priv_key` as input and returns the signature `sig` as output.

Since the input `msg` may be sensor data generated *just in time*, in the context of our work, it is called an *online* input. Furthermore, any output or intermediate variable that is control- or data-dependent on the *online* input must be computed *just in time*. In contrast, results that do not depend on the *online* input may be computed *ahead of time*.

2.2.1 The Original Program

Figure 4 shows the definitions of the two subroutines invoked by `wots()`. The subroutine `sign()` in Line 7 takes `msg` and `priv_key` as input and returns `sig` as output. While `msg` is an *online* input, `priv_key` is computed by the subroutine `gen_key()`. In this sense, `sign()` depends on `gen_key()`.

The subroutine `gen_key()` does not have any input, and thus does not depend on any other subroutine. More importantly, it does not depend on any *online* input. Thus, `gen_key()` may be executed ahead of time, e.g., whenever ambient energy is available to the harvester. It means that both `priv_key` and `pub_key` may be computed ahead of time. These precomputed keys may be saved to non-volatile memory as *coupons*, and later used by `sign()` to encrypt the *online* input `msg`.

Although the subroutine `sign()` partially depends on the *online* input `msg`, and thus cannot be executed ahead of time *in its entirety*, a significant part of the function body can still be executed ahead of time. Specifically, the subroutine `gen_random()` does not depend on the *online* input at all, and the subroutine `memcpy()` depends only on `rand` computed by `gen_random()`; thus, both subroutines can be computed ahead of time.

```

1  gen_key(priv_key, pub_key) {
2      gen_random(priv_key, PRIV_KEY_SIZE);
3      sha256_init(&keyHash);
4      sha256_update(&keyHash, priv_key, PRIV_KEY_SIZE);
5      sha256_final(&keyHash, pub_key);
6  }
7  sign(msg, priv_key, sig) {
8      gen_random(rand, SHA_BLK_SIZE);
9      memcpy(sig, rand, SHA_BLK_SIZE);
10     message_digest(digest_bits, sig, msg);
11     gen_sig(sig, priv_key, digest_bits);
12 }

```

■ **Figure 4** Definitions of the subroutines used by the W-OTS routine.

```

1  wots_precom(msg, pub_key, sig) {
2      gen_key(priv_key, pub_key);
3      //NVM-Store <priv_key, pub_key> to coupon pool;
4      sign_precom(msg, priv_key, sig);
5  }
6  wots_online(msg, pub_key, sig) {
7      //NVM-Load <priv_key, pub_key> from coupon pool;
8      sign_online(msg, priv_key, sig);
9  }
10 gen_key(priv_key, pub_key) {
11     gen_random(priv_key, PRIV_KEY_SIZE);
12     sha256_init(&keyHash);
13     sha256_update(&keyHash, priv_key, PRIV_KEY_SIZE);
14     sha256_final(&keyHash, pub_key);
15 }
16 sign_precom(msg, priv_key, sig) {
17     gen_random(rand, SHA_BLK_SIZE);
18     memcpy(sig, rand, SHA_BLK_SIZE);
19     //NVM-Store <sig> to coupon pool;
20 }
21 sign_online(msg, priv_key, sig) {
22     //NVM-Load <sig> from coupon pool;
23     message_digest(digest_bits, sig, msg);
24     gen_sig(sig, priv_key, digest_bits);
25 }

```

■ **Figure 5** Conceptually, the program may be divided into two parts (*precom* and *online*).

If we continue this analysis by going down the chain of function calls, we may identify more precomputation opportunities, e.g., instructions inside subroutines `message_digest()` and `gen_sig()`. In our proposed method, this process of systematically identifying these precomputation opportunities is automated, based on static program analysis techniques.

2.2.2 Dividing into Two Parts

Based on the precomputation opportunities identified by static program analysis, the original program may be divided into two parts: the precomputation (*precom*) part and the online computation (*online*) part, as shown by Figure 5.

Specifically, top-level routine `wots()` is divided into `wots_precom()` and `wots_online()`. The subroutine `wots_precom()` may be invoked ahead of time, since it does not depend on the *online* input `msg` at all. After invoking `gen_key()` to compute the public and private keys, denoted `priv_key` and `pub_key`, it stores them in non-volatile memory (Line 3). Then, it invokes `sign_precom()` defined in Line 16, to compute the signature `sig`, before storing it in non-volatile memory (Line 19).

The subroutine `wots_online()` must be invoked just in time, since it depends on the *online* input `msg`. This subroutine first loads the precomputed keys `priv_key` and `pub_key` from non-volatile memory (Line 7) and then invokes `sign_online()` defined in Line 21. Inside `sign_online()`, the precomputed signature `sig` is loaded from non-volatile memory (Line 22) and then used together with `msg` and `priv_key` to compute the final version of the signature `sig` (Lines 23-24).

According to our experimental evaluation (presented in Section 7), on low-power devices such as MSP430, this kind of precomputation can reduce the energy cost of running W-OTS to 42.89% of the original cost. In other words, it is more than 2.3X reduction. Thus, with the same amount of electricity used to run the original W-OTS program once, now, we can run the optimized W-OTS program 2.3 times.

2.2.3 Challenges in Optimization

Just because an instruction may be precomputed (i.e., it does not depend on any *online* input) does not mean that it is beneficial to do so, since precomputation does not always reduce the energy cost. Depending on the hardware platform, it is possible for the cost of storing and retrieving the precomputed result to outweigh the benefit.

For example, in Line 18 of Figure 5, if we choose to precompute `memcpy()` inside the subroutine `sign()`, the energy cost of loading the precomputed coupon `sig` from non-volatile memory may be slightly higher than the energy needed to execute `memcpy()` directly. If that is the case, precomputation should be avoided.

In general, whether precomputation is beneficial or not depends on both the software and the hardware. Consider the characteristics of volatile and non-volatile memory used in MSP430FR5969 [24] as an example. According to the hardware data-sheet, at the clock frequency of 8 MHz, the energy per clock cycle is 0.33 nJ for volatile memory, but is 0.42 nJ for non-volatile memory. This kind of information must be considered during optimization.

Furthermore, decisions on which instructions to precompute cannot be made in isolation, since many of these instructions are dependent on each other according to the control and data flows of the program. Therefore, we must consider all of the intra- and inter-procedural control- and data-flow dependencies in the program while performing the optimization.

These are the reasons why we propose the constraint based method. By first formulating it as a constraint satisfiability problem and then solving the problem using an off-the-shelf SMT solver, we are able to optimally partition the program into the precomputation part and the online computation part, while satisfying a variety of requirements coming from the hardware platform as well as the software program.

2.2.4 The Optimized Program

To keep the size of the optimized program small, we do not actually divide the program into two parts as shown by Figure 5. Instead, we keep the two parts in a single program, and try to retain the original control and data flows of the program as much as possible.

Figure 6 illustrates our method by showing the optimized program for the original program in Figure 4. Our method adds two parameters, `precom_flag` and `online_flag`, to represent the following three use cases:

- When $\langle \text{precom_flag}, \text{online_flag} \rangle = \langle \text{true}, \text{false} \rangle$, it does precomputation.
- When $\langle \text{precom_flag}, \text{online_flag} \rangle = \langle \text{false}, \text{true} \rangle$, it does online computation.
- When $\langle \text{precom_flag}, \text{online_flag} \rangle = \langle \text{true}, \text{true} \rangle$, it acts as the original program.


```

1  wots_trans(msg, pub_key, sig, precom_flag, online_flag) {
2    if (precom_flag == true)
3      gen_key(priv_key, pub_key);
4    if (!online_flag)
5      //NVM-Store <priv_key, pub_key> to coupon pool;
6    if (!precom_flag)
7      //NVM-Load <priv_key, pub_key> from coupon pool;
8    sign_trans(msg, priv_key, sig, precom_flag, online_flag);
9  }
10 sign_trans(msg, priv_key, sig, precom_flag, online_flag) {
11   if (precom_flag == true) {
12     gen_random(rand, SHA_BLK_SIZE);
13     memcpy(sig, rand, SHA_BLK_SIZE);
14   }
15   if (!online_flag)
16     //NVM-Store <sig> to coupon pool;
17   if (!precom_flag)
18     //NVM-Load <sig> from coupon pool;
19   if (online_flag == true) {
20     message_digest(digest_bits, sig, msg);
21     gen_sig(sig, priv_key, digest_bits);
22   }
23 }

```

■ **Figure 6** Merging the two parts into a single optimized W-OTS routine.

```

1  __interrupt void ISR(void) {
2    if (!msg_ready) {
3      if (ambient_energy_available)
4        wots_trans(NULL, pub_key, sig, true, false); //precom (part 1)
5    }
6    else {
7      if (!ambient_energy_available)
8        wots_trans(msg, pub_key, sig, false, true); //online (part 2)
9      else
10       wots_trans(msg, pub_key, sig, true, true); //combined (part 1 + part 2)
11     //Send the pair <pub_key,sig> to verifier;
12   }
13 }

```

■ **Figure 7** Different scenarios for invoking the optimized W-OTS routine.

Compared to the original program in Figure 4, the only difference in Figure 6 is the addition of two flags as input parameters of some of the subroutines, together with the if-conditions that indicate whether a code block should be executed during the precomputation step or during the online computation step.

Figure 7 shows how the optimized program may be invoked by the interrupt service routine. Unlike what is shown in Figure 3, here, precomputation is performed when `msg` is not available but ambient energy is available (Line 4). When `msg` is available, it depends on whether ambient energy is still available. If ambient energy is not available, then online computation is performed (Line 8). However, if ambient energy is available, operations that access non-volatile memory will be skipped, which makes `wots_trans()` behaves exactly the same as the original program (Line 10).

3 Overview of Our Method

We first present our top-level procedure and then outline the main technical challenges.

3.1 The Top-Level Procedure

Algorithm 1 shows our top-level procedure. The input consists of the original program (P), the *online* input (OI) of the program, and the system constraint (C). The output is the optimized program (P').

■ **Algorithm 1** The top-level procedure of our method.

```

input : original program  $P$ , online input  $OI$ , system constraint  $C$ 
output : optimized program  $P'$ 
1  $PDG \leftarrow \text{ConstructPDG}(P)$ ;
2  $preSet \leftarrow \text{IdentifyPreSet}(P, PDG, OI)$ ;
3  $preSet^* \leftarrow \text{OptimizePreSet}(preSet, PDG, C)$ ;
4  $P' \leftarrow \text{Transform}(P, PDG, preSet^*)$ ;
5 return  $P'$ 

```

For the running example in Figure 3, where the entry function is `wots()`, the online input is $OI = \{\text{msg}\}$, since `msg` is the only input value that must be ready at run time. C consists of a set of platform-dependent requirements, e.g., the size of non-volatile memory used to store coupons must be limited to ≤ 256 KB.

In Algorithm 1, our method first constructs a program dependency graph (PDG) for the program P . Then, our method uses the PDG and the set of variables in the *online* input OI to compute $preSet$, which is the set of instructions in P that may be precomputed. Next, it computes $preSet^*$, which is a subset of $preSet$ that represents the optimal solution to the constraint satisfiability problem. Finally, our method transforms the program P to a new program P' based on the information stored in both PDG and $preSet^*$.

Before presenting the detailed algorithms inside the subroutines `IdentifyPreSet()`, `OptimizePreSet()` and `Transform()`, we point out the main technical challenges.

3.2 The Technical Challenges

The first challenge, related to the subroutine `IdentifyPreSet()`, is the complex nature of the program dependency analysis. In Figures 3 and 4, for example, we observe that the subroutine `sign()` depends on `gen_key()`; furthermore, the subroutine `gen_sig()` invoked by `sign()` depends on `gen_key()`. It means that we must consider not only dependencies of instructions within each subroutine, but also dependencies between subroutines.

Moreover, since we aim to transform individual functions of the original program without changing the overall function call structure, each function must be analyzed in all of its calling contexts, to figure out how the function body should be optimized. In Figure 4, for example, it means that since `gen_random()` is called by both `gen_key()` and `sign()`, we must consider both calling contexts.

The second challenge, related to the subroutine `OptimizePreSet()`, is optimizing the precomputation policy while satisfying a variety of system constraints. Given $preSet$ (which is the set of instructions that may be computed), we need to identify a proper subset. For the MSP430 family of microcontroller units, a limiting factor may be the capacity of non-volatile

memory, only part of which may be dedicated to coupon storage. In general, this is a non-linear optimization problem, e.g., the storage cost may not increase linearly, or even monotonically, as more instructions are added to the precomputation set.

In Figure 4, for example, the cost of precomputing only Lines 2-4 is $size(\text{priv_key}) + size(\text{keyHash})$, where $size()$ denotes the size of non-volatile memory for storing the value. However, the cost of precomputing Lines 2-5 is $size(\text{priv_key}) + size(\text{pub_key})$, because keyHash no longer needs to be stored in non-volatile memory. Since $size(\text{pub_key})$ is much smaller than $size(\text{keyHash})$ in the W-OTS example, this means that precomputing one more line actually decreases the overall storage cost.

The third challenge, related to the subroutine `Transform()`, is the difficulty in preserving functional equivalence while allowing the program to change its execution order and data flow. For example, if we want to precompute Line 2 and Line 8 in Figure 4, we must modify the program to ensure that the original execution order (Line l_3 executed before Line l_8) changes to the new execution order (l_8 executed before l_3); at the same time, we must ensure that the original data flow $\text{priv_key}(l_2) - l_3, l_4, l_5 - l_8$ changes to $\text{priv_key}(l_2) - l_8 - l_3, l_4, l_5$. While doing so for this particular example may seem easy, in general, maintaining functional equivalence during such program transformation can be challenging.

4 Identifying the Precomputation Set

In this section, we present our method for computing $preSet$, as shown in Algorithm 2. It takes the program P , the program dependency graph PDG , and the *online* input OI as parameters, and return $preSet$ as output.

Algorithm 2 The subroutine `IdentifyPreSet` (P, PDG, OI).

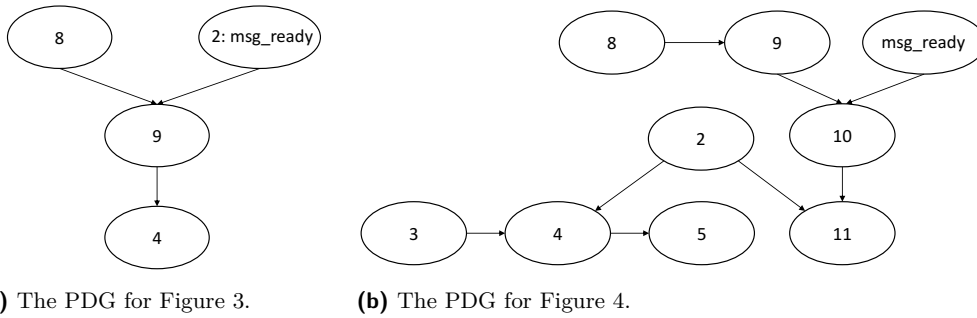
```

1 Let  $pred(inst)$  be a predecessor node of instruction  $inst$  in the  $PDG$ 
2  $preSet \leftarrow \{\text{elementary instructions in } P\} \cup (\{\text{input parameters of } P\} \setminus OI)$ 
3 while  $\exists inst \in preSet$  and  $pred(inst) \notin preSet$  do
4   | remove  $inst$  from  $preSet$ 
5 end
6 return  $preSet$ 

```

Recall that $preSet$ is the set of instructions in P that may be computed ahead of time. Internally, our method computes $preSet$ in two steps. The first step is identifying the interprocedural dependencies related to the online input OI . These dependencies will be captured by function such as $pred(inst)$, $preds(inst)$, and $succs(inst)$, which returns the predecessor, set of predecessors, and set of successors of an instruction $inst$, respectively. The second step is leveraging these dependencies to compute the instructions in $preSet$.

In Algorithm 2, initially, $preSet$ consists of all the *elementary instructions* and input parameters of P , except for the ones in OI . Variables in OI are excluded because they are the *online* variables. Here, an *elementary instruction* means that during our analysis the instruction will be treated as a whole. First, non-function-call instructions are elementary instructions. Second, when an instruction invokes a function call, whether it is elementary depends on how many times the function is called. If the function is called only once, it is not treated as an elementary instruction; instead, we enter the function body to try to identify more precomputation opportunities. But if the function is called from multiple sites, we treat each call as an elementary instruction, meaning that we do not enter the function body



■ **Figure 8** The program dependency graphs (PDGs) of the example W-OTS program. Here, each node represents an instruction, and the number is the instruction’s line number in the program.

to explore further. This is a reasonable compromise since, when a function is called from multiple sites, the function body often implements some basic computation, e.g., generating a random number, and there is no need to split it further.

4.1 Inter-Procedural Dependencies

To identify the maximum set of instructions in *PreSet* using Algorithm 2, we need the dependencies associated with the online input *OI*. These dependencies are more complex than what are typically available in the compiler. For example, by default, LLVM provides the control- and data-dependencies between instructions only within each function. However, we need to know dependencies not only within each function, but also between functions.

To identify inter-procedural dependencies, we first compute a PDG for each function, together with a call graph that represents the *caller-callee* relations of all functions in the program. We also extend LLVM to add the ability to determine whether a function call may change the content of a function parameter passed by reference or the value of a global variable. This is accomplished by traversing paths in the call graph and analyzing all of the functions involved in the path.

Next, we analyze the inter-procedural dependencies in a bottom-up fashion, according to the function call graph. Consider the example of the following two functions: `fun1(arg1)` and `fun2(arg2, arg3)`, where the input parameter `arg1` of `fun1()` depends on the output parameter `arg2` of `fun2()`. Assume that `arg3` is also an output parameter of `fun2()`.

Assume that inside the function `fun2()` there is an instruction *I* that computes the value of `arg2`. Furthermore, inside `fun1()` there is an instruction *I'* that computes the value of `arg1`. While all intra-procedural dependencies may be computed in isolation, we must combine them to identify the inter-procedural dependencies, such as the dependency between *I'* of `fun1()` and *I* of `fun2()`.

Figure 8 shows a more concrete example, where the PDGs are constructed for the code snippets in Figures 3 and 4. Consider the edge $2 \rightarrow 11$ in Figure 8 (b), which represents the dependency between the instruction at Line 2 and the instruction at Line 11 of the program in Figure 4. It means the input parameter `priv_key` used by `sign()` at Line 11 comes from the output parameter `priv_key` of `gen_key()` at Line 2.

With the inter-procedural dependencies, we can define the notion of a *predecessor*, denoted by *pred()*. For example, in Figure 8 (b), due to the edge $2 \rightarrow 11$, we say that the instruction at Line 2 is a predecessor of the instruction at Line 11 inside the program shown by Figure 4.

4.2 Iteratively Computing *preSet*

Using the notion of a *predecessor* of an instruction *inst*, denoted $pred(inst)$, our method computes the *preSet* according to the while-loop in Algorithm 2.

It starts with all elementary instructions and input parameters that are not in *OI*. Then, it removes any instruction (*inst*) that has a predecessor $pred(inst)$ not in *preSet*. There are two possible reasons why $pred(inst)$ is not in *preSet*: either it is in *OI*, or during the previous iteration, it has been removed from *preSet*. Thus, it is a fixed-point computation.

The correctness of the fixed-point computation can be understood as follows: By definition, the instruction *inst* depends on its predecessor $pred(inst)$. If $pred(inst) \notin preSet$, meaning the predecessor instruction cannot be precomputed, then the instruction *inst* itself cannot be precomputed either.

As an example, consider the instructions of W-OTS in Figure 4. For ease of presentation, we use l_i to represent the instruction at Line *i*, and we treat all instructions in this program as elementary instructions. Initially, we have $preSet = \{l_2 - l_5, l_8 - l_{11}\}$.

Next, we check if any of these instructions should be removed, based on the *predecessor* relation shown in Figure 8. The instruction l_{10} should be removed, since its predecessor (`msg_ready`) is not in *preSet*. Thus, we remove l_{10} from *preSet*.

The removal of l_{10} leads to the removal of l_{11} during the next iteration, since l_{10} is the predecessor of l_{11} . If l_{11} cannot be precomputed, then l_{10} cannot be precomputed either.

Thus, in the end, we have $preSet = \{l_2 - l_5, l_8 - l_9\}$.

► **Theorem 1.** *Our method for computing *preSet* is sound in that, for all $inst \in preSet$, there is guarantee that the instruction (*inst*) can indeed be computed ahead of time.*

Proof. An instruction *inst* remains in *preSet* only if all of its predecessors are also in *preSet*. As long as the inter-procedural dependencies represented by the PDGs are an over-approximation of the actual dependencies, the *preSet* is guaranteed to be an under-approximation of the set of instructions that may be computed ahead of time.

The reason why it is an under-approximation because $pred(inst)$ is an over-approximation of the predecessors. Whenever $pred(inst) \notin preSet$, Algorithm 2 removes *inst* from *preSet*.

The reason why $pred(inst)$ is an over-approximation is due to the nature of PDG-based analysis techniques. Refer to Horwitz et al. [22] and Reps et al. [42] for more information. ◀

4.2.1 Handling Loops

Similar to all other PDG-based analysis techniques [22, 42], our method has no problem in handling software code with loops. In most of the practical cases, computing the *predecessor* is straightforward. For example, the function call `sign()` at Line 9 in Figure 3 requires `msg` and `priv_key` to be available. These dependencies are due to data flow represented by the *definition-use* correspondence.

However, there are cases where definitions and uses do not have one-to-one mapping. For example, in Figure 9, the variable `i` used at Line 7 may be defined at either Line 2 or Line 5. In the context of data-flow analysis, the definition at Line 5 does not *kill* the definition at Line 2. Therefore, it may or may not be necessary to precompute Line 3-6 in order to precompute Line 7, for example, if `CNT[len-1] != 0xff`.

Since our method is designed to be sound, to ensure that the optimized program is correct for all input values, it is allowed to *first* over-approximate the predecessor relation, and *then* conservatively assume that an instruction can be precomputed *only if* all of its predecessors can be precomputed.

```

1 void increment_CNT(BYTE *CNT, int len){
2     int i = len;
3     while ((i > 0) && (CNT[i-1] == 0xff)){
4         CNT[i-1] = 0;
5         i--;
6     }
7     if (i) {
8         CNT[i-1]++;
9     }
10 }

```

■ **Figure 9** Code snippet taken from the benchmark program named AES-CTR.

5 Optimizing the Precomputation Set

While all instructions in $preSet$ have been identified at this moment, it may not be beneficial to compute all of them ahead of time. In this section, we present our method for computing an optimal subset $preSet^* \subseteq preSet$. This is implemented in $OptimizePreSet(preSet, PDG, C)$, where C is the system constraint. Besides the characteristics of the hardware platform, such as the size of non-volatile memory, it also includes the characteristics of the software program, such as how often the encrypted sensor data must be transmitted to the remote server.

5.1 The Motivation

We use an example to illustrate the complex nature of the optimization problem, which in turn motivates our development of the constraint based solution.

Consider the W-OTS program in Figure 4 and its PDGs in Figure 8 (b). According to Algorithm 2, $preSet = \{l_2 - l_5, l_8 - l_9\}$. Since these instructions do not depend on the *online* input `msg`, in theory, they may be precomputed *as many times as possible*. However, due to the storage capacity, in practice, the number has to be bounded.

Let \mathcal{S}_i be a subset of $preSet$, called a precomputation choice, and m_i be the maximum number of times that \mathcal{S}_i may be precomputed. Since each time \mathcal{S}_i produces an intermediate result, or *coupon*, we also call m_i the *coupon count* (number of copies of this particular coupon). Let $NVM(\mathcal{S}_i)$ be the storage cost for this coupon, and \maxNVM be the storage capacity of the entire device. We use the maximal allowed NVM size to avoid the potential risk of running out of NVM. One precomputation choice for the running example is represented by $\mathcal{S}_1 = \{l_2\}$, where $m_1 \leq \maxNVM/NVM(\mathcal{S}_1)$. That is, the coupon count m_1 is bounded only by the storage capacity.

Below are some other precomputation choices:

$$\begin{aligned} \mathcal{S}_2 &= \{l_2 - l_5, l_8\}, \text{ where } m_2 \leq \maxNVM/NVM(\mathcal{S}_2) \\ \mathcal{S}_3 &= \{l_2 - l_5, l_8 - l_9\}, \text{ where } m_3 \leq \maxNVM/NVM(\mathcal{S}_3) \\ &\dots \end{aligned}$$

Let $n = |preSet|$, the number of precomputation choices is $\sum_{i=1}^n \binom{n}{i}$. Since it causes combinatorial explosion, we cannot afford to enumerate them to decide which one is optimal.

The number of precomputation choices can be even higher than $\sum_{i=1}^n \binom{n}{i}$. For example, when $\mathcal{S}_{4a} = \{l_2 - l_5\}$ and $\mathcal{S}_{4b} = \{l_2 - l_5, l_8 - l_9\}$, if we allow the coupon counts m_{4a} and m_{4b} to have different values, they would be bounded only by the constraint $m_{4a} \times NVM(\mathcal{S}_{4a}) + m_{4b} \times NVM(\mathcal{S}_{4b}) \leq \maxNVM$. This leads to another combinatorial explosion.

While making a precomputation choice, we cannot consider instructions in isolation, since they may be dependent on each other. For example, precomputing one instruction may require precomputing another instruction. Recall that in the example program shown in Figure 4, we cannot precompute l_5 without precomputing l_4 , because there is dependency from l_4 to l_5 . In other words, $l_4 = \text{pred}(l_5)$.

All these challenges motivate us to define the constraint satisfiability problem, which allows us to consider all of the selected instructions as a whole, together with a variety of system constraints. Specifically, it allows us to consider the coupon count (m_i) and the coupon size $\text{NVM}(\mathcal{S}_i)$ for each subset $\mathcal{S}_i \subseteq \text{preSet}$, together with system constraints such as the capacity of non-volatile memory used to store coupons computed by different instructions, and the inter-procedural dependencies between these chosen instructions.

5.2 The Problem Statement

Our goal is to compute the optimal subset, denoted $\mathcal{S}^* \subseteq \text{preSet}$, that satisfies the system constraint. For ease of presentation, assume that \mathcal{S} represents a precomputation choice, while $V(\mathcal{S})$ represents the value (or benefit) of precomputing \mathcal{S} , and $C(\mathcal{S})$ represents the cost of precomputing \mathcal{S} . The optimization problem is defined formally as follows:

$$\mathcal{S}^* = \underset{\mathcal{S} \subseteq \text{preSet}}{\text{argmax}} V(\mathcal{S}) \quad \text{subject to} \quad C(\mathcal{S}) \leq \text{maxNVM} \quad (5.2)$$

In other words, the optimal subset is the subset \mathcal{S} that maximize the value $V(\mathcal{S})$ while keeping the cost $C(\mathcal{S})$ under control. Recall that explicitly enumerating solutions would lead to combinatorial explosion. Thus, we encode them symbolically using a set of logical constraints and solve these constraints using an off-the-shelf SMT solver.

One advantage of the *constraint based approach* is flexibility in modeling various tradeoffs. While it is easy to compute the coupon size or the coupon count individually, finding the right combination may be hard due to the fact that they are inter-dependent.

Another advantage of our approach is flexibility in modeling the chain of influence; that is, precomputing one instruction (e.g., l_4 of `gen_key` in Figure 4) may require precomputing another instruction (e.g., l_3).

Yet another advantage is the ability to bound the total cost of storing coupons from different instructions. As mentioned earlier, precomputing more instructions may not always increase the storage cost. In Figure 4, if we precompute $l_3 - l_4$ but not l_5 , we need to store both `pub_key` and `keyHash`, the latter of which is an array of 108 bytes; but if we precompute $l_3 - l_5$, we only need to store `pub_key`, which is an array of 32 bytes.

5.3 Defining the Value and Cost Functions

First, we define the energy saving (*value*) and storage overhead (*cost*).

5.3.1 Value

Since the value of precomputing one instruction may depend on which other instructions are precomputed, we can only define it based on which other instructions are chosen. Since an instruction *inst* may be precomputed only if all its *predecessors* are precomputed, we define the value of precomputing *inst* based on the predecessor relation.

16:16 Constraint Based Compiler Optimization for Energy Harvesting Applications

Let \mathcal{S} be the set of chosen instructions, and $v(inst \mid \mathcal{S})$ be the *value* of precomputing $inst$ in the presence of \mathcal{S} . We have

$$v(inst \mid \mathcal{S}) = \begin{cases} E(inst) & \text{if } preds(inst) \subseteq \mathcal{S} \\ -\infty & \text{otherwise} \end{cases}$$

Here, $E(inst)$ is the energy saved by precomputing $inst$, and $preds(inst)$ is the set of all predecessors of $inst$ in the PDG. We use the large value $-\infty$ to avoid precomputing $inst$ before all of its predecessors in $preds(inst)$ are precomputed.

With the values of precomputing individual instructions, we define the *value* of precomputing the entire set \mathcal{S} as follows:

$$V(\mathcal{S}) = \sum_{inst \in \mathcal{S}} v(inst \mid \mathcal{S}).$$

For the example in Figures 4 and 8 (b), we have $V(\{l_2\}) = E(l_2)$. We also have $V(\{l_2, l_5\}) = -\infty$ since l_5 cannot be selected when its predecessors $l_3 - l_4$ are not selected.

5.3.2 Cost

Unlike the value $v(inst)$, which depends only on the predecessors of $inst$, the cost of precomputing $inst$ depends also on its *successors* in the PDG.

Let \mathcal{S} be the set of chosen instructions, and $c(inst \mid \mathcal{S})$ be the *cost* of precomputing $inst$ in the presence of \mathcal{S} . In Figure 4, for instance, we have

$$c(l_3 \mid \mathcal{S}) = \begin{cases} 0 & \text{if } l_4, l_5 \in \mathcal{S} \\ \text{NVM}(\text{keyHash}) & \text{otherwise} \end{cases}$$

and

$$c(l_4 \mid \mathcal{S}) = \begin{cases} 0 & \text{if } l_2, l_3 \in \mathcal{S} \\ +\infty & \text{otherwise} \end{cases}$$

That is, if $l_3 - l_5$ are selected, we do not need to store `keyHash`; but if $l_4 - l_5$ are not selected, we need to store `keyHash`. Thus, the cost of precomputing l_3 depends on if $(l_4 - l_5)$ are selected. Here, the large value $+\infty$ is used to avoid selecting instructions whose predecessors in the PDG are not selected.

With the costs of precomputing individual instructions, we define the cost of precomputing the entire set \mathcal{S} as follows:

$$C(\mathcal{S}) = \sum_{inst \in \mathcal{S}} c(inst \mid \mathcal{S}).$$

5.4 Symbolic Encoding of the Constraints

We construct an SMT formula $\Psi = \Phi_{Dep} \wedge \Phi_{Value} \wedge \Phi_{Cost}$, where the subformula Φ_{Dep} captures the dependencies that we have computed in the previous section, Φ_{Value} captures the value constraint, and Φ_{Cost} captures the cost constraint. Thus, a satisfying assignment to Ψ corresponds to $\mathcal{S}^* \subseteq preSet$.

5.4.1 Dependency Constraint

Φ_{Dep} encodes the dependency relations captured by edges of the inter-procedural PDG. Specifically, for each dependency edge (n_1, n_2) , we add a Boolean constraint $(\neg n_2 \vee n_1)$, where n_1 and n_2 are Boolean variables indicating whether these nodes are precomputed, and the constraint means that, if n_2 is true, then n_1 must also be true. Therefore, n_2 being precomputed implies that n_1 is also precomputed. Then, all these individual constraints are conjoined to form Φ_{Dep} . As an example, consider the PDG in Figure 8 (b): the dependency constraints include $(\neg l_4 \vee l_3) \wedge (\neg l_4 \vee l_2) \wedge (\neg l_5 \vee l_4) \wedge (\neg l_9 \vee l_8)$.

5.4.2 Value Constraint

Φ_{Value} encodes the value of precomputing each instruction. Since Φ_{Dep} already guarantees that an instruction is precomputed only if all its predecessors (as in the PDG) are precomputed, the encoding becomes straightforward. That is, if $inst$ is selected, then $v(inst) = E(inst)$; otherwise $v(inst) = 0$. The total value of precomputing the set of instructions in $preSet$ is simply the sum of all the individual values. In Figure 4, the value of precomputing each instruction l_i , where $i = 2, 3, \dots, 5, 8, 9$, would be $v(l_i) = (l_i ? E(l_i) : 0)$ and the total would be $V(\mathcal{S}) = \sum v(l_i)$.

5.4.3 Cost Constraint

Φ_{Cost} encodes the cost of precomputing the chosen instructions. Recall that the cost of precomputing $inst$ depends on not only if its predecessors are precomputed but also if its successors are precomputed. Since Φ_{Dep} guarantees to select the predecessors whenever $inst$ is selected, here we only need to deal with the set of successors, denoted $succs(inst)$.

In general, precomputing $inst$ increases storage cost only when its result (coupon) is used by some of the successors in the online computation step; otherwise, there is no need to save the coupon. For example, the cost of precomputing l_3 in Figure 4 is zero if instructions in $succs(l_3) = \{l_4, l_5\}$ are also precomputed.

For the entire program shown in Figure 4, the cost constraint would be

$$\begin{aligned} & (c(l_2) = (\neg l_2 \vee l_3 \wedge l_4 \wedge l_5 \wedge l_{send}) ? 0 : \text{NVM}[\text{priv_key}]) \wedge \\ & (c(l_3) = (\neg l_3 \vee l_4) ? 0 : \text{NVM}[\text{keyHash}]) \wedge \\ & (c(l_4) = (\neg l_4 \vee l_5) ? 0 : \text{NVM}[\text{keyHash}]) \wedge \\ & (c(l_5) = \neg l_5 ? 0 : \text{NVM}[\text{pub_key}]) \wedge \\ & (c(l_8) = (\neg l_8 \vee l_9) ? 0 : \text{NVM}[\text{rand}]) \wedge \\ & (c(l_9) = \neg l_9 ? 0 : \text{NVM}[\text{sig}]) \\ & (C(\mathcal{S}) = c(l_2) + c(l_3) + c(l_4) + c(l_5) + c(l_8) + c(l_9)) \wedge \\ & (C(\mathcal{S}) \leq \text{maxNVM}) \end{aligned}$$

With proper definitions of the cost and value functions, our constraint based method can also handle other optimization metrics.

5.5 Solving the Constraints

After constructing the entire SMT formula Ψ , we solve it using the Z3 SMT solver [11]. Specifically, we use Z3's `optimize` interface iteratively to search for the optimal solution. This is done by insisting that the total value $V(\mathcal{S})$ shown in Equation (5.2) is greater than a given constant value; then, we find the maximum constant by gradually increasing the value of the constant as long as Z3 can still find a satisfying solution.

6 Transforming the Program

We now explain the subroutine $\text{Transform}(P, PDG, preSet^*)$, which transforms the original program P to a new program P' to implement $preSet^*$. Recall that in Figure 6, we gave an example of such a transformed program for W-OTS. There are two important properties of the program P' : (1) it retains the overall function call structure in P and (2) it changes the body of each function to implement both the precomputation and online computation steps.

6.1 The Terminology

For each function f in the program P , we must separate the precomputation instructions from the online computation instructions. This leads to a partition of the program to segments, $\{S_1, \tilde{S}_2, S_3, \tilde{S}_4, \dots\}$, where S_i represents a precomputation segment and \tilde{S}_j represents an online computation segment. A *segment* is a maximal set of instructions that may execute continuously during precomputation or online computation.

Consider an example program $P = \{S_1, \tilde{S}_2, S_3, \tilde{S}_4\}$ whose original execution order is $S_1 \rightarrow \tilde{S}_2 \rightarrow S_3 \rightarrow \tilde{S}_4$. In the transformed program P' , however, the execution order must be changed to $S_1 \rightarrow S_3 \rightarrow \tilde{S}_2 \rightarrow \tilde{S}_4$. In general, changes in the execution order lead to changes in the data flow.

Before discussing changes in the data flow, we define the terminology.

- Let $def(x)$ be an instruction that defines the value of variable x , and $use(x)$ be an instruction that uses the value. The two instructions may form a *def-use* pair.
- Given two segments S_i and \tilde{S}_j , where $def(x) \in S_i$ and $use(x) \in \tilde{S}_j$, we represent the data-flow edge (or def-use pair) as $\langle S_i, \tilde{S}_j \rangle(x)$.
- Let $Val[x, S_i]$ denote the value of x at the end of executing the segment S_i .
- A variable x is *live* at a program location p if its value is used before it is defined again along *some* path from p to the program exit.

6.2 The Problem

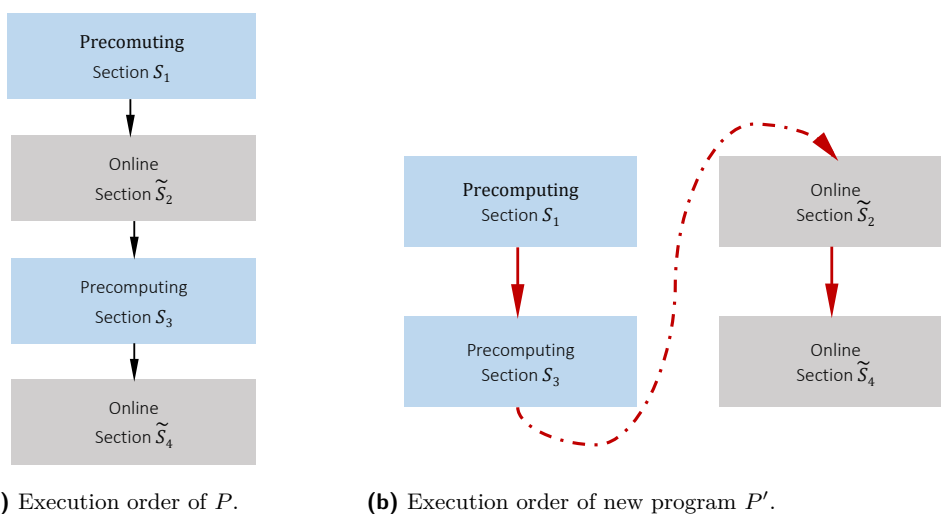
Now, we show an example where changes in the execution order bring unexpected changes of the data flow.

► **Example 6.1.** In program $P = \{S_1, \tilde{S}_2, S_3, \tilde{S}_4\}$, assume that $def_1(x) \in S_1$, $def_2(x) \in \tilde{S}_2$, $use(x) \in \tilde{S}_4$. Due to the execution order, the def-use chain contains only $def_2(x)$ and $use(x)$, meaning the value of x used in \tilde{S}_4 should be from $def_2(x)$.

In the original execution order $S_1 \rightarrow \tilde{S}_2 \rightarrow S_3 \rightarrow \tilde{S}_4$, the value $Val[x, S_3]$ comes from $def_2(x)$, and the variable x is live in S_3 , since $Val[x, S_3]$ will be used in \tilde{S}_4 .

In the new program, however, since the execution order is changed to $S_1 \rightarrow S_3 \rightarrow \tilde{S}_2 \rightarrow \tilde{S}_4$, without our intervention, the value $Val[x, S_3]$ would come from $def_1(x)$, and the variable x would *no longer* be live in S_3 . Such unexpected changes of the data flow may change the semantics of the program. This is illustrated by Figure 10.

In general, it can be challenging to preserve the data flow while allowing change of the execution order. While the technique of *checkpointing* has been used in intermittent computing systems [32, 48, 35], it cannot solve our problem because checkpointing does not involve splitting a program into two parts and then executing the two parts in an interleaved order. For the program in Example 6.1, specifically, checkpointing techniques would have failed to preserve the data flow.



■ **Figure 10** Difference in execution order means P and P' are no longer functionally equivalent.

To understand why checkpointing would fail, consider the fact that variable x is *live* at the end of \tilde{S}_2 , at the end of S_3 , and at the start of \tilde{S}_4 . Checkpointing would insert $\text{nvm_ST}(Val[x, \tilde{S}_2])$ at the end of \tilde{S}_2 and insert $\text{nvm_ST}(Val[x, S_3])$ at the end of S_3 . It would also insert $\text{nvm_LD}(Val[x, \tilde{S}_2])$ and $\text{nvm_LD}(Val[x, S_3])$ at the start of \tilde{S}_4 .

When executing P' ($S_1 \rightarrow S_3 \rightarrow \tilde{S}_2 \rightarrow \tilde{S}_4$), $\text{nvm_LD}(Val[x, S_3])$ would over-write $\text{nvm_LD}(Val[x, \tilde{S}_2])$; thus, the value of x used in \tilde{S}_4 would be $Val(x, S_3) = \text{def}_1(x)$. However, in the original program, the value of x used in \tilde{S}_4 is $\text{def}_2(x)$.

The fundamental reason why *checkpointing* techniques are ill-suited for our project is that the *liveness* property of a program variable, which forms the theoretical foundation of checkpointing techniques, is not preserved by the split of a program into the precomputation and online computation parts. Thus, instead of relying on the *liveness* property, our method relies on the *def-use* relations.

6.3 The Baseline Method

We first present the baseline method using the *def-use* relations, and then present the optimized method in the next subsection.

Since we treat each segment as an atomic unit during transformation, we only need to consider the *def-use* relations between segments. Thus, whenever two segments have *def-use* relations, there can only be three scenarios:

- (I) $\langle S_i, S_j \rangle$, meaning both are precomputation segments;
- (II) $\langle S_i, \tilde{S}_j \rangle$, meaning S_i is a precomputation and \tilde{S}_j is an online computation; and
- (III) $\langle \tilde{S}_i, \tilde{S}_j \rangle$, meaning both are online computation segments.

The fourth scenario, $\langle \tilde{S}_i, S_j \rangle$, is impossible due to our method for computing *preSet*.

In other words, a *use* in a precomputation segment always comes from a *definition* in a precomputation segment, whereas a *use* in an online computation segment may come from a *definition* in a precomputation or an online computation segment.

Furthermore, it suffices to handle only type (II) case $\langle S_i, \tilde{S}_j \rangle$, because for the other two cases, the value *can be* propagated directly between the two segments of the same type.

To maintain the *def-use* chains between precomputation and online computation segments in the type (II) case, we must insert nvm_LD and nvm_ST instructions at the proper *def* and *use* locations.

Thus, our baseline method can be summarized as follows: For each data-flow edge $\langle S_i, \tilde{S}_j \rangle(x)$, we insert $\text{nvm_ST}(Val[x, S_i])$ at the end of S_i , and insert $\text{nvm_LD}(Val[x, S_i])$ at the start of \tilde{S}_j .

Recall the scenario shown in Example 6.1, where the *def-use* chain contains only $\text{def}_2(x)$ and $\text{use}(x)$. According to our baseline method, no NVM operation needs to be added, since the *def-use* is of the type (III). The value of x used in \tilde{S}_4 comes directly from $\text{def}_2(x)$.

6.4 The Optimized Method

Now, we present an optimization to avoid *redundant* NVM operations inserted by the baseline method. To understand why some of the NVM operations inserted by our baseline method may be redundant, consider the following example.

► **Example 6.2.** In $\{S_1, \tilde{S}_2, S_3, \tilde{S}_4\}$, assume that $\text{def}(x) \in S_1$, $\text{use}_1(x) \in \tilde{S}_2$, $\text{use}_2(x) \in \tilde{S}_4$, and the *def-use* chain contains both $\text{def}(x)\text{-use}_1(x)$ and $\text{def}(x)\text{-use}_2(x)$. Our baseline method would insert

- $\text{nvm_ST}(Val[x, S_1])$ after S_1 (twice);
- $\text{nvm_LD}(Val[x, S_1])$ before \tilde{S}_2 ;
- $\text{nvm_LD}(Val[x, S_1])$ before \tilde{S}_4 .

However, executing $\text{nvm_LD}(Val[x, S_1])$ before \tilde{S}_4 is redundant because the value of x can be propagated directly from \tilde{S}_2 .

To avoid the redundant operations, we should insert nvm_LD of a $\text{def}(x)$ at the start of the earliest online computation segment where $\text{def}(x)$ is available. For the program in Example 6.2, the earliest segment is \tilde{S}_2 , which means we should insert $\text{nvm_LD}(Val[x, S_1])$ right before \tilde{S}_2 .

Thus, our optimized method can be summarized as follows: For each data-flow edge $\langle S_i, \tilde{S}_j \rangle(x)$ that we have not inserted $\text{nvm_ST}(Val[x, S_i])$ after S_i , insert $\text{nvm_ST}(Val[x, S_i])$ after S_i and insert $\text{nvm_LD}(Val[x, S_i])$ before \tilde{S}_{i+1} .

To understand the benefit of this optimization, let us compare the data flows of the following two programs. If, for example, in the original program, $Val[x, S_i]$ is available (and not killed) in the range

$$\text{end}[S_i] \rightarrow \tilde{S}_{i+1} \rightarrow S_{i+2} \rightarrow \tilde{S}_{i+3} \rightarrow \dots \quad (1)$$

and in the transformed program, $Val[x, S_i]$ is available (and not killed) in the range

$$\text{end}[S_i] \rightarrow S_{i+2} \rightarrow S_{i+4} \rightarrow S_{i+6} \rightarrow \dots \quad (2)$$

and $\text{nvm_LD } Val[x, S_i]$ has been inserted before \tilde{S}_{i+1} in the transformed program, the loaded value will also be available in the entire range

$$\tilde{S}_{i+1} \rightarrow \tilde{S}_{i+3} \rightarrow \tilde{S}_{i+5} \rightarrow \tilde{S}_{i+7} \rightarrow \dots \quad (3)$$

Therefore, we can avoid the other (redundant) nvm_LD operations before $\tilde{S}_{i+3} \dots \tilde{S}_{i+7}$.

6.5 The Transformation Algorithm

To sum up, our optimized method for transforming each function f of the original program based on preSet^* is presented in Algorithm 3.

Our method first partitions the instructions in function f to precomputation segments $\{S_i\}$ and online computation segments $\{\tilde{S}_j\}$. Next, it inserts *if-condition* to each segment using the two flags, to differentiate the three use cases. Finally, for each data-flow edge

■ **Algorithm 3** Transforming a function f in program P based on $preSet^*$.

```

1 Partition  $f$  into segments  $\{S_i\}$  and segments  $\{\tilde{S}_j\}$ ;
2 Add if-condition to each segment using precom_flag and online_flag;
3 foreach data-flow edge denoted  $\langle S_i, \tilde{S}_j \rangle(x)$  do
4   | if there is no nvm_ST(Val[x, Si]) after segment  $S_i$  then
5   |   | Add nvm_ST(Val[x, Si]) after  $S_i$ ;
6   |   | Add nvm_LD(Val[x, Si]) before  $\tilde{S}_{i+1}$ ;
7   | end
8 end

```

$\langle S_i, \tilde{S}_j \rangle(x)$, it insert NVM operations to store the value of variable x computed in S_i (the coupon) at the end of segment S_i .

While in the baseline method, the coupon is loaded from NVM at the start of \tilde{S}_j , in the optimized method, it is loaded at the start of the online computation segment \tilde{S}_{i+1} . Loading the coupon earlier provides the opportunity to eliminate many redundant NVM operations.

7 Experiments

We have implemented our method in a software tool, named COUPONMAKER, which builds upon the LLVM compiler platform [29] and the Z3 SMT solver [11]. We leverage LLVM to parse the C code of the original program, conduct inter-procedural dependency analysis and implement the semantic-preserving transformation. We use Z3 to solve the constraint satisfiability subproblems. In total, our implementation adds 1,852 lines of C++ code.

Our tool generates the LLVM bit-code of the optimized program as output, which in turn is compiled to machine code for the MSP430 MCU. To evaluate the performance of the optimized program, we use the cycle-accurate emulator MSPSim [39]. Specifically, we use MSPSim to compute the latency and energy consumption of the optimized program, and compare them with the latency and energy consumption of the original program.

7.1 Benchmarks

We evaluated COUPONMAKER on 26 benchmark programs, which are C programs implementing lightweight cryptographic protocols. In total, they have 31,113 lines of C code. Table 1 shows the statistics, where Columns 1-3 show the name, category, and source of each program, and Column 4 shows the number of lines of code (LoC).

The benchmark programs fall into two groups. The first group consists of programs that compute one-time signatures (W-OTS and Lamport) and the second group consists of programs that implement block-ciphers (e.g., AES and Camellia). A one-time signature scheme allows a message to be signed using a fresh key pair. Since any fresh key pair may work for any message, it is possible to precompute many key pairs and store them as coupons for future use. A block cipher divides a message into fixed-size blocks and then encrypts each block. For example, AES-CTR encrypts each block by first encrypting a counter value and then XOR-ing it with the plaintext to generate the ciphertext. The precomputing function is responsible for encrypting the counter value. Since there are multiple blocks, different counter values need to be encrypted. For each of the eight block-cipher programs, we also configure it in three different modes, marked by suffixes -OFB, -CFB, and -CTR, respectively.

■ **Table 1** Statistics of the benchmark programs.

Name	Category	Source	LoC
W-OTS	One-time signature	Merkle signature [40]	1,062
Lamport	One-time signature	Lamport signature[28]	339
AES	Block cipher	OpenSSL[37]	1,572
Camellia	Block cipher	OpenSSL[37]	708
DES	Block cipher	avr-crypto-lib[6]	1,277
Blowfish	Block cipher	OpenSSL[30]	1,112
skipjack	Block cipher	avr-crypto-lib[6]	475
GOST	Block cipher	OpenSSL[37]	357
SEED	Block cipher	OpenSSL[30]	476
CAST128	Block cipher	OpenSSL[31]	963

Our experiments were conducted on a computer with 2 GHz Intel Core i5 CPU and 16 GB memory. These experiments were designed to answer the following questions:

- Is COUPONMAKER efficient in optimizing the benchmark programs?
- Are the optimized programs better than the original programs in terms of both energy efficiency and latency?

7.2 Performance of the Optimization Tool

Table 2 shows the results of evaluating the optimization tool. Column 1 shows the benchmark name. Column 2 shows the total running time in seconds. Column 3 shows the size of *preSet*, which is the set of instructions that may be precomputed. Columns 4-5 compare the size of the original and optimized programs, where the size is measured in the number of bytes of the LLVM bit-code. Columns 6-8 show the details of the coupons stored in non-volatile memory, including the number of coupons, and the total bytes, and whether the coupons may be precomputed multiple times (copies).

Specifically, ∞ in the last column means the coupons may be precomputed an unlimited number of times, while 1 means they may be precomputed only once.

For programs that compute one-time signatures (W-OTS and Lamport), a theoretically unbounded number of signatures (coupons) may be precomputed. For block-cipher programs in the -OFB mode, the ciphertext of the first block may also be precomputed as many times as possible (after the first block becomes available), and in the -CNT mode, the counter *CNT* may be incremented as many times as possible and then pre-encrypted for future use.

For block-cipher programs in the -CFB mode, however, precomputation can only be done once per block, i.e., after the current block arrives.

The results show that our method is able to analyze, optimize, and transform all benchmark programs quickly. The total running time is limited to a few seconds. Moreover, the size of the program before and after optimization changes moderately. Furthermore, the number and size of precomputed coupons are significant for all programs.

7.3 Performance of the Optimized Programs

Table 3 shows the result of evaluating the performance of the optimized programs. These results were obtained using the MSPSim tool for MSP430FR599x [24]. Since MSPSim requires the programs to be executed under concrete test inputs, for one-time signature programs (W-OTS and Lamport), we obtain the test inputs by signing a fixed-length message; for block-cipher programs, we obtain the test inputs by encrypting sensor data that represent a sequence of temperature measurements.

■ **Table 2** Performance of the analysis tool COUPONMAKER.

Name	Time (s)	PreSet Size	Program Size		Coupon Size		
			orig.	opti.	num	bytes	copies
W-OTS	5.26	1,632	16,116	21,704	3	1,152	∞
Lamport	4.08	1,000	14,268	19,116	2	512	∞
AES-OFB	3.35	3,964	52,636	57,984	1	16	∞
AES-CFB	3.62	3,964	56,162	56,168	1	16	1
AES-CTR	3.73	4,064	53,164	58,584	1	16	∞
Camellia-OFB	3.37	1,412	20,228	25,276	1	16	∞
Camellia-CFB	3.30	1,412	20,696	25,788	1	16	1
Camellia-CTR	3.89	1,460	24,964	29,984	1	16	∞
DES-OFB	3.11	2,072	26,384	26,496	1	8	∞
DES-CFB	3.14	2,072	26,432	26,644	1	8	1
DES-CTR	3.05	2,112	26,896	27,556	1	8	∞
Blowfish-OFB	3.38	1,196	16,200	21,308	1	8	∞
Blowfish-CFB	3.27	1,196	16,180	21,288	1	8	1
Blowfish-CTR	3.70	1,242	16,636	21,724	1	8	∞
skipjack-OFB	3.09	1,896	34,452	39,552	1	8	∞
skipjack-CFB	3.26	1,896	34,404	39,536	1	8	1
skipjack-CTR	3.32	1,940	34,864	40,008	1	8	∞
GOST-OFB	2.79	596	12,508	17,504	1	8	∞
GOST-CFB	3.16	596	12,492	17,484	1	8	1
GOST-CTR	3.01	844	12,952	17,984	1	8	∞
SEED-OFB	2.67	196	31,120	36,384	1	8	∞
SEED-CFB	2.68	196	31,100	36,368	1	8	1
SEED-CTR	3.11	340	31,564	36,852	1	8	∞
CAST128-OFB	2.49	352	46,628	51,748	1	8	∞
CAST128-CFB	2.74	352	46,608	51,732	1	8	1
CAST128-CTR	3.00	396	47,064	52,228	1	8	∞

In the result table, Column 1 shows the benchmark name. Column 2 shows the energy (μJ) consumed by the original program. Columns 3-4 show the energy (μJ) consumed by the optimized program, which is divided into the precomputing and online steps. Recall that in energy-harvesting applications, energy reported in the $E(\text{pre})$ column is considered to be free. Thus, the ratio in Column 5 represents the actual performance improvement.

The results show that the optimized programs significantly outperform the original programs in terms of energy efficiency. The improvement ranges from 2.3X to 36.7X. We also compared the latency of the original and optimized programs and observed a similar improvement; we omit the result table due to space limit. Overall, these results show that our method is effective in reducing the latency and energy cost.

7.4 Impact of the Precomputation Policy

Finally, we evaluate the impact of precomputation policy by computing the energy saving per unit use of non-volatile memory storage, measured by $qf = (E(\text{ori}) - E(\text{on})) / \text{Size}(\text{coupon})$, where qf stands for quality factor. The results are shown in Figure 11, where the x -axis is the index of the array of benchmark programs and the y -axis is the quality factors (qf) achieved by the baseline and optimized methods for program transformation (Section 6).

In this figure, blue bars (*optimal*) correspond to the optimized precomputation policy (preSet^*), while orange bars (*baseline*) corresponds to the initial precomputation policy (preSet). Here, a higher qf value corresponds to a better result. Overall, the optimized precomputation policy leads to significantly better results.

■ **Table 3** Evaluating reduction in energy cost on MSP430.

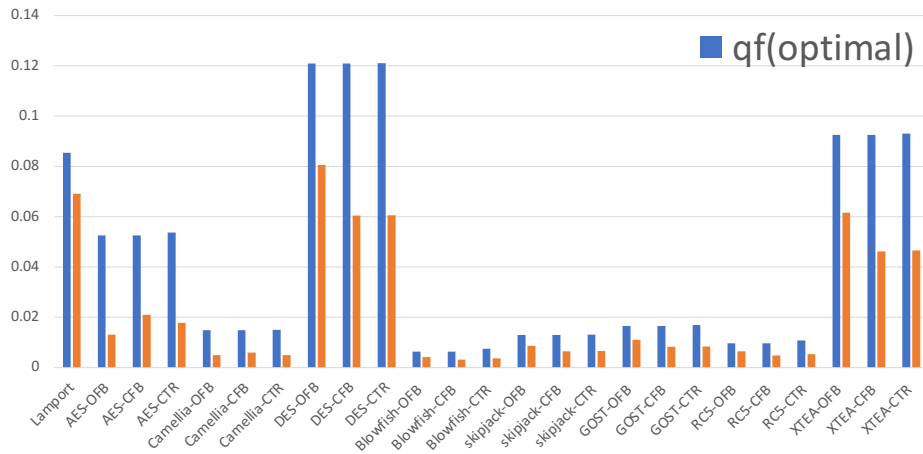
Name	Original Program	Optimized Program		Improvement
	$E(ori)$	free $E(pre)$	$E(on)$	$E(ori)/E(on)$
W-OTS	115565.43	56114.99	49576.70	2.3X
Lamport	355.91	287.31	82.71	4.3X
AES-OFB	89.06	87.96	4.85	18.4X
AES-CFB	90.67	87.96	6.46	14.0X
AES-CTR	89.23	88.15	3.36	26.5X
Camellia-OFB	28.66	27.56	4.85	5.9X
Camellia-CFB	30.27	27.56	6.46	4.7X
Camellia-CTR	28.84	27.75	4.87	5.9X
DES-OFB	198.84	197.87	5.42	36.7X
DES-CFB	200.56	197.88	7.14	28.1X
DES-CTR	199.18	198.25	5.45	36.6X
Blowfish-OFB	15.63	14.66	5.43	2.9X
Blowfish-CFB	17.35	14.66	7.14	2.4X
Blowfish-CTR	15.97	12.64	4.01	4.0X
skipjack-OFB	26.16	25.20	5.42	4.8X
skipjack-CFB	29.33	25.20	8.58	3.4X
skipjack-CTR	26.73	26.06	5.71	4.7X
GOST-OFB	29.10	29.01	2.59	11.3X
GOST-CFB	29.83	29.01	3.32	9.0X
GOST-CTR	29.65	29.61	2.62	11.3X
SEED-OFB	20.32	19.21	4.85	4.2X
SEED-CFB	21.92	19.21	6.45	3.4X
SEED-CTR	20.49	17.64	3.22	6.4X
CAST128-OFB	164.89	161.86	16.89	9.8X
CAST128-CFB	170.24	161.86	22.24	7.7X
CAST128-CTR	165.95	163.05	16.99	9.8X

For W-OTS, qf (optimal) is also significantly higher than qf (baseline). However, the qf values for W-OTS are not included in the figure, to avoid making the rest of the bar chart less readable. This is because W-OTS takes several orders-of-magnitude more clock cycles than the other programs, and thus has a much higher qf value.

8 Related Work

While prior work has shown the feasibility of optimizing energy-harvesting applications using precomputation [47], optimization is performed manually; to the best of our knowledge, this is the first automated optimization method. Compared to Suslowicz et al. [47], in particular, our method can complete all of the optimization work with comparable performance. Moreover, our method can support additional constraints for optimization, which the manual method cannot deal with easily. Since our method is designed to preserve the original program semantics, it is not meant for scenarios where the underlying algorithms are intended to be rewritten according to some mathematical rules [4, 5] – automation for such transformation is beyond the scope of this work.

Our method differs from the large number of intermittent computing techniques aimed to improve general-purpose systems with a strong and yet unstable power supply; these techniques [43, 35, 32, 48] focus on recovering from power loss using *checkpointing*, avoiding the costly register accesses, or reducing the cost for loop-heavy programs [18, 17]. There are also techniques for robustly supporting peripherals [46, 36]. However, none of them considers the scenario where ambient energy source is ample but the computing device is idle, let alone leveraging precomputation to reduce the energy cost.



■ **Figure 11** The impact of the precomputation policy on performance improvement. Here, baseline corresponds to *preSet* and optimal corresponds to *preSet**.

There are also techniques for programming transiently-powered computers with both volatile and non-volatile memory, for example, by leveraging the application’s memory access patterns to manually optimize data placement [9, 32, 34], or mapping of code sections to either volatile or non-volatile memory [25] based on where the optimal energy consumption could be achieved. There are also efficient checkpointing techniques [21, 1] for CPUs with fully non-volatile main memory. However, none of them focuses on automated program optimization based on precomputation.

Constraint solving based techniques are widely used for program verification, repair and optimization. For example, they have been used to debug concurrent software [27, 23] and optimize the quality of embedded software [13]. They have also been used to mitigate side-channel vulnerabilities [49, 19, 52, 50], including power side-channel leaks [54, 51]. However, power side-channel mitigation focuses on eliminating *tiny fluctuations* in power consumption that are also *secret-dependent* [14], instead of reducing the power consumption itself.

While our focus in this work is on optimizing software for energy-harvesting applications, the underlying ideas may be applied to other applications of similar nature, e.g., precomputation for Trusted Authority (TA) in the context of multi-party computation (multi-party learning and predicting[53, 16]). Since the application domain is significantly different, to deal with software used in such applications, our LLVM based implementation may need to be updated accordingly – we leave this for future work.

9 Conclusion

We have presented a constraint based method for optimizing the energy efficiency of software code running on devices powered by electricity harvested from the environment. Our method is sound and fully automated. It relies on static program analysis to identify instructions that may be precomputed, constraint solving to compute an optimal subset, and compiler transformation to generate the new software code. Our experimental evaluation on a large number of benchmark programs shows that the proposed method can handle all of the benchmark programs quickly, and the optimized programs significantly outperform the original programs in terms of both energy efficiency and latency.

References

- 1 Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In Jian-Jia Chen and Aviral Shrivastava, editors, *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 70–81. ACM, 2019. doi:10.1145/3316482.3326357.
- 2 Saad Ahmed, Muhammad Nawaz, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Demystifying energy consumption dynamics in transiently powered computers. *ACM Trans. Embed. Comput. Syst.*, 19(6):47:1–47:25, 2020. doi:10.1145/3391893.
- 3 James Allen, Matthew Forshaw, and Nigel Thomas. Towards an extensible and scalable energy harvesting wireless sensor network simulation framework. In Walter Binder, Vittorio Cortellessa, Anne Koziolok, Evgenia Smirni, and Meikel Poess, editors, *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 39–42. ACM, 2017. doi:10.1145/3053600.3053610.
- 4 Giuseppe Ateniese, Giuseppe Bianchi, Angelo Caposelle, and Chiara Petrioli. Low-cost standard signatures in wireless sensor networks: a case for reviving pre-computation techniques? In *Network and Distributed System Security Symposium*, 2013.
- 5 Giuseppe Ateniese, Giuseppe Bianchi, Angelo T Caposelle, Chiara Petrioli, and Dora Spenza. Low-cost standard signatures for energy-harvesting wireless sensor networks. *ACM Transactions on Embedded Computing Systems*, 16(3):64, 2017.
- 6 The avr-crypto-lib software package. <https://github.com/cantora/avr-crypto-lib>. Accessed: 2019-09-26.
- 7 Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1):15–18, 2015.
- 8 Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In Tom Conte and Yuanyuan Zhou, editors, *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 577–589. ACM, 2016. doi:10.1145/2872362.2872409.
- 9 Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In Eelco Visser and Yannis Smaragdakis, editors, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 514–530. ACM, 2016. doi:10.1145/2983990.2983995.
- 10 Riccardo Dall’Ora, Usman Raza, Davide Brunelli, and Gian Pietro Picco. SenseH: From simulation to deployment of energy harvesting wireless sensor networks. In *IEEE 39th Conference on Local Computer Networks, Edmonton, AB, Canada, 8-11 September, 2014 – Workshop Proceedings*, pages 566–573. IEEE Computer Society, 2014. doi:10.1109/LCNW.2014.6927704.
- 11 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- 12 The tiny Dutch startup solving the IoT industry’s battery problem. <https://sifted.eu/articles/nowi-dutch-startup-solving-iot-battery-problem/>. Accessed: 2020-08-04.
- 13 Hassan Eldib and Chao Wang. An SMT based method for optimizing arithmetic computations in embedded software code. In *International Conference on Formal Methods in Computer-Aided Design*, pages 129–136. IEEE, 2013.
- 14 Hassan Eldib, Chao Wang, Mostafa M. I. Taha, and Patrick Schaumont. Quantitative masking strength: Quantifying the power side-channel resistance of software code. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34(10):1558–1568, 2015.

- 15 Joakim Eriksson, Fredrik Österlind, Thiemo Voigt, Niclas Finne, Shahid Raza, Nicolas Tsiftes, and Adam Dunkels. Accurate power profiling of sensor networks with the COOJA/MSPSim simulator. In *IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 1060–1061, 2009.
- 16 Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.
- 17 Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213. ACM, 2019.
- 18 Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. MANIC: A vector-dataflow architecture for ultra-low-power embedded systems. In *IEEE/ACM International Symposium on Microarchitecture*, pages 670–684, 2019.
- 19 Shengjian Guo, Meng Wu, and Chao Wang. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 377–388. ACM, 2018.
- 20 Josiah D. Hester, Timothy Scott, and Jacob Sorber. Ekho: realistic and repeatable experimentation for tiny energy-harvesting sensors. In Ákos Lédeczi, Prabal Dutta, and Chenyang Lu, editors, *ACM Conference on Embedded Network Sensor Systems*, pages 1–15. ACM, 2014. doi:10.1145/2668332.2668336.
- 21 Matthew Hicks. Clank: Architectural support for intermittent computation. In *International Symposium on Computer Architecture*, pages 228–240. ACM, 2017. doi:10.1145/3079856.3080238.
- 22 Susan Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In Tony Montgomery, Lori A. Clarke, and Carlo Ghezzi, editors, *International Conference on Software Engineering, Melbourne, Australia, May 11-15, 1992*, pages 392–411, 1992.
- 23 Zunchen Huang and Chao Wang. Symbolic predictive cache analysis for out-of-order execution. In *International Conference on Fundamental Approaches to Software Engineering*, pages 163–183. Springer, 2022.
- 24 Texas Instrument. MSP430FR599x Technical Documentation. URL: <https://www.ti.com/product/MSP430FR5994>.
- 25 Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. Energy-aware memory mapping for hybrid FRAM-SRAM mcus in intermittently-powered iot devices. *ACM Trans. Embed. Comput. Syst.*, 16(3):65:1–65:23, 2017. doi:10.1145/2983628.
- 26 Mustafa Emre Karagozler, Ivan Poupyrev, Gary K Fedder, and Yuri Suzuki. Paper generators: harvesting energy from touching, rubbing and sliding. In *ACM symposium on User interface software and technology*, pages 23–30, 2013.
- 27 Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, pages 165–176. ACM, 2015.
- 28 The lamport_signature software package. https://github.com/detomastah/lamport_signature. Accessed: 2019-09-26.
- 29 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization*, page 75, 2004.
- 30 The Libgcrypt software package. <https://gnupg.org/software/libgcrypt/index.html>. Accessed: 2019-09-26.
- 31 The Libmcrypt software package. <https://github.com/tugrul/libmcp-crypt-gyp/tree/master>. Accessed: 2019-09-26.

- 32 Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 50(6):575–585, 2015.
- 33 Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *IEEE International Symposium on High Performance Computer Architecture*, pages 526–537, 2015.
- 34 Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA):96:1–96:30, 2017. doi:10.1145/3133920.
- 35 Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 129–144, 2018.
- 36 Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1101–1116, 2019.
- 37 Shorter Merkle Signatures. <https://www.openssl.org>. Accessed: 2019-09-26.
- 38 Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *IEEE International Conference on Pervasive Computing and Communications*, pages 216–224, 2013.
- 39 The MSP430 emulator. <https://github.com/contiki-ng/mspsim>.
- 40 OpenSSL. <https://www.openssl.org>. Accessed: 2019-09-26.
- 41 Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ACM SIGARCH Computer Architecture News*, pages 159–170, 2011.
- 42 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23–25, 1995*, pages 49–61, 1995.
- 43 Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1085–1100, 2019.
- 44 Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement*, 57(11):2608–2615, 2008.
- 45 Claude E. Shannon. Communication theory of secrecy systems. *Bell Syst. Tech. J.*, 28(4):656–715, 1949.
- 46 Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/O dependent idempotence bugs in intermittent systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):183, 2019.
- 47 Charles Suslowicz, Archanaa S Krishnan, and Patrick Schaumont. Optimizing cryptography in energy harvesting applications. In *Proceedings of the Workshop on Attacks and Solutions in Hardware Security*, pages 17–26. ACM, 2017.
- 48 Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 17–32, 2016.
- 49 Chao Wang and Patrick Schaumont. Security by compilation: an automated approach to comprehensive side-channel resistance. *ACM SIGLOG News*, 4(2):76–89, 2017.
- 50 Jingbo Wang, Chunggha Sung, Mukund Raghothaman, and Chao Wang. Data-driven synthesis of provably sound side channel analyses. In *International Conference on Software Engineering*, pages 810–822. IEEE, 2021.

- 51 Jingbo Wang, Chungha Sung, and Chao Wang. Mitigating power side channels during compilation. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 590–601. ACM, 2019.
- 52 Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In Frank Tip and Eric Bodden, editors, *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26. ACM, 2018.
- 53 Jiawei Yuan and Shucheng Yu. Privacy preserving back-propagation neural network learning made practical with cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):212–221, 2013.
- 54 Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. SCInfer: Refinement-based verification of software countermeasures against side-channel attacks. In *International Conference on Computer Aided Verification*, pages 157–177. Springer, 2018.

Restrictable Variants: A Simple and Practical Alternative to Extensible Variants

Magnus Madsen ✉ 

Department of Computer Science, Aarhus University, Denmark

Jonathan Lindegaard Starup ✉ 

Department of Computer Science, Aarhus University, Denmark

Matthew Lutze ✉ 

Department of Computer Science, Aarhus University, Denmark

Abstract

We propose restrictable variants as a simple and practical alternative to extensible variants. Restrictable variants combine nominal and structural typing: a restrictable variant is an algebraic data type indexed by a type-level set formula that captures its set of active labels. We introduce new pattern-matching constructs that allows programmers to write functions that only match on a subset of variants, i.e., pattern-matches may be non-exhaustive. We then present a type system for restrictable variants which ensures that such non-exhaustive matches cannot get stuck at runtime.

An essential feature of restrictable variants is that the type system can capture structure-preserving transformations: specifically the introduction and elimination of variants. This property is important for writing reusable functions, yet many row-based extensible variant systems lack it.

In this paper, we present a calculus with restrictable variants, two partial pattern-matching constructs, and a type system that ensures progress and preservation. The type system extends Hindley-Milner with restrictable variants and supports type inference with an extension of Algorithm W with Boolean unification. We implement restrictable variants as an extension of the Flix programming language and conduct a few case studies to illustrate their practical usefulness.

2012 ACM Subject Classification Theory of computation → Program semantics

Keywords and phrases restrictable variants, extensible variants, refinement types, Boolean unification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.17

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.12>

1 Introduction

“Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.”
– Antoine de Saint-Exupéry

In functional programming, algebraic data types and pattern matching have been hugely successful. So successful that many non-functional mainstream programming languages, including Kotlin and Rust have also adopted them. While algebraic data types, i.e. sum and variant types, are widely used, their cousins *extensible variants* and *extensible records* are far less available. Extensible variants and records, based on row-polymorphic type systems, have been known for several decades [11, 16, 37]. Yet one has to look far to find usable implementations. OCaml does not support extensible records, but does support a form of extensible variants as a “language extension”, but this implementation is far less powerful than simple row-polymorphic systems. PureScript supports extensible records, but not extensible variants.¹ Elm had support for extensible records, but this feature was removed.² We speculate that there are at least a few reasons for this lack of support: (i) lack of real (or perceived) use cases, (ii) implementation difficulty, and (iii) hitting the “right” expressiveness.

¹ <https://github.com/purescript/documentation/blob/master/language/Records.md>

² <https://github.com/elm/compiler/issues/985>



© Magnus Madsen, Jonathan Lindegaard Starup, and Matthew Lutze;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 17; pp. 17:1–17:27



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



17:2 Restrictable Variants

To expand on (iii), we believe we have identified a major practical weakness in existing row-based extensible variant systems. We illustrate the problem with an example: In a compiler pipeline, we can view each compiler phase as a function, and the whole compiler as the composition of these functions. For example, we might have:

```
let compile = ... >> typecheck >> lambdalift >> codegen
```

where `>>` is forward function composition. The `lambdalift` phase performs closure conversion and lambda lifting which is required before we can generate machine code. Concretely, we can imagine that the `lambdalift` function replaces `Lambda` expressions with `Closure` expressions in the abstract syntax tree. We say that `lambdalift` *introduces* the `Closure` variant and *eliminates* the `Lambda` variant. Importantly, we must run `lambdalift` before we can run `codegen`.

As compiler writers, it would be very useful if we could type check the abstract syntax trees produced by `lambdalift`. That is, we would like to write:

```
let compile = ... >> typecheck >> lambdalift >> typecheck >> codegen
```

This requires us to extend the `typecheck` function to handle `Closure` expressions, but that is simple; type checking them is similar to type checking lambdas.

We might think that the above scenario can be programmed with row-based extensible variants, but, unfortunately, this is not the case. The problem is the following: The `codegen` phase *cannot* handle abstract syntax trees unless they have been closure-converted and lambda-lifted, i.e. unless the `Lambda` expression has been eliminated. But after the second call to `typecheck`, a row-based system loses the knowledge that the `Lambda` variant has been eliminated, hence the above program does not type check.

We call this phenomenon the *co-domain problem* for extensible variants:

The Co-Domain Problem: Type systems with extensible variants based on row polymorphism are unable to precisely capture the *introduction* and *elimination* of variants in pattern-matches. (We expand on the details in Section 5.)

To overcome this issue, we propose *restrictable variants*. A restrictable variant is a sum type indexed by a type-level set formula that over-approximates the “active” set of labels of the sum. We can think of a restrictable variant as a form of refinement type [10, 36] where the type-level index refines the possible labels of an expression of that type. In this way, restrictable variants combine nominal and structural typing. With restrictable variants, programmers can write *one* data type definition and reuse it in different contexts. This is in contrast to the standard functional programming approach of writing multiple, but similar, data types definitions or using a purely structural type system.

In this paper, we introduce restrictable variants and a new partial pattern-matching construct which comes in two flavors: `choose` and `choose*`. The `choose` expression permits a non-exhaustive pattern-match on a restrictable variant where only some variants are handled. The `choose*` expression goes further and enables programmers to write structure-preserving transformations which are captured at the type level. Specifically, we can precisely capture the *introduction* or *elimination* of variants. This overcomes the co-domain problem that was outlined above. We propose a type system for restrictable variants which is an extension of Hindley-Milner, supports complete type inference, and ensures that programs with partial pattern-matches (i.e. with `choose` and `choose*`) cannot get stuck.

We compare the expressiveness of restrictable variants to other existing systems, including row-based extensible variants [11, 16, 33, 37], row theories [30], occurrence typing [7], and relational nullable types [25]. We find that many of these systems are significantly more expressive (and sometimes more complex) than restrictable variants, yet most cannot express

the simple programming patterns that we use in our case studies. We think that restrictable variants (like row-based systems) have simple types that will be understandable by ordinary programmers and which will work well in practice. It is also our hope that restrictable variants can serve as inspiration for new and more sophisticated type systems that can handle the use cases we present.

The ideas in this paper are simple, but as far as we can tell, they have not yet been explored in the literature, and we believe they solve real problems. While we present *restrictable variants* as an alternative to *extensible variants*, we have found that it is natural to combine restrictable variants with *extensible records* (a point we return to in Section 6).

We implement restrictable variants as an extension of the Flix programming language. We discuss how the implementation supports complete type inference as a natural extension of Algorithm W. The two key ideas are: (i) a type rule, for the `choose*` expression, which relates the type-level index of the “input” (scrutinee expression) to the “output” (result expression), and (ii) a formulation of the type rule as a *set equation* which is solvable by Boolean unification in the algebra of sets.

We use the implementation to conduct a case study of a few programs that use restrictable variants. The first case study models Boolean formulas and is used as a running example throughout the paper. The second case study combines the `Option`, `List`, and `Nel` (non-empty list) data types into one restrictable variant. The third case study shows how to combine restrictable variants with extensible records. The case studies demonstrate that programming with restrictable variants is simple and valuable.

In summary, the contributions of this paper are:

- **(Restrictable Variants)** We present *restrictable variants*: a simple alternative to extensible variants. Restrictable variants offer a new point in the design space with different trade-offs from existing systems. Moreover, restrictable variants solve the function composition problem for row-based extensible variants.
- **(Type System)** We present a type system for restrictable variants. We prove the standard progress and preservation theorems. The type system ensures that a program with partial pattern-matches (the `choose` and `choose*` expressions) cannot get stuck.
- **(Implementation)** We implement restrictable variants as an extension of the Flix programming language. We discuss how the type system supports type inference via an extension of Algorithm W with Boolean unification.
- **(Expressiveness)** We compare the expressiveness of restrictable variants to other systems. We observe that restrictable variants are simple, yet they support reasonable use cases that cannot be handled by many other systems, in particular those based on row polymorphism.
- **(Case Study)** We present a case study of a few programs that use restrictable variants. The case study shows that (a) restrictable variants are useful and (b) capture real-world programming patterns.

This paper is organized as follows: In Section 2 we present restrictable variants and motivate their use with several examples. In Section 3 we present a type system for restrictable variants based on type-level set formulas. In Section 4 we discuss our implementation of restrictable variants and show how to implement type inference. In Section 5 we compare the expressiveness of restrictable variants to other systems, including row-based extensible variants. In Section 6 we present a case study on the use of restrictable variants. In Section 7 we present related work and in Section 8 we conclude the paper.

2 Motivation

We motivate restrictable variants with several examples. We begin with a simple example to build intuition. Next, we move on to a more realistic example of modeling Boolean formulas, which we use as a running example throughout the rest of the paper. We show many types, but, of course, the point is that they can be inferred. All examples are runnable in our extension of Flix.

► **Example 1 (Restrictable Variant).** We can define a *restrictable variant* data type:

```
enum Color[s] {
  case Red
  case Green
  case Blue
}
```

The `Color` type is *indexed* by a type variable `s` that ranges over the *labels* of the algebraic data type. The labels of the `Color` data type are: `Red`, `Green`, and `Blue`. The index is a set formula that captures which variants of the data type may be present. For example:

$$\text{Color}\{\{\}\} = \emptyset \quad \text{Color}\{\{\text{Red}, \text{Blue}\}\} = \{\text{Red}, \text{Blue}\} \quad \text{Color}[\text{Green}^c] = \{\text{Red}, \text{Blue}\}$$

We can also have richer indices where a free variable is involved. For example:

$$\text{Color}[s - \text{Red}] \subseteq \{\text{Green}, \text{Blue}\} \quad \{\text{Green}, \text{Blue}\} \subseteq \text{Color}[(s - \text{Red}) + (s^c)] \subseteq \{\text{Red}, \text{Green}, \text{Blue}\}$$

where s is a free variable. In full generality, the index is a *type-level set formula* whose valuations capture which variants of the data type may be present. As the examples show, there are many equivalent set formulas. For example, Green^c is equivalent to the set $\{\text{Red}, \text{Blue}\}$. The set formulas may also contain variables, a fact that becomes important when we consider pattern-matches on restrictable variants.

We can write a function that only operates on some colors of a restrictable variant:

```
def isWarm(c: Color[{Red, Blue}]): Bool = choose c {
  case Red => true
  case Blue => false
}
```

Here the type of `isWarm`, which can be fully inferred, captures that the function can accept any color which is either `Red` or `Blue`. Specifically, the type system ensures that it is a compile-time type error to call `isWarm` with the color `Green`.

This example demonstrates a key feature of the proposed type system: We can write pattern-matches that are non-exhaustive and have the type system ensure that a function like `isWarm` is never invoked with a value that is not handled.

With some intuition in place, we now move on to our running example: a restrictable variant that models Boolean formulas. We use Boolean formulas since they are well-known and they are sufficient to illustrate several key features of our system. We want to stress that the following ideas scale to more complex data types, e.g. abstract syntax trees, as we shall discuss in Section 6.

A remark on notation: In Flix source code we shall write $\neg s$ for S^c , $s_1 + s_2$ for $S_1 \cup S_2$, and $s_1 \& s_2$ for $S_1 \cap S_2$. We use these symbols because they are in ASCII and are reminiscent of the bitwise operators. In the formal treatment of the calculus and its type system (Section 3), we will use the standard math symbols.

► **Example 2 (Variant Restriction).** We can define a restrictable variant for Boolean formulas:

```
enum Expr[s] {
  case Var(Int32)
  case Cst(Bool)
  case Not(Expr[s])
  case Or(Expr[s], Expr[s])
  case And(Expr[s], Expr[s])
  case Xor(Expr[s], Expr[s])
}
```

We can write a function which reduces *closed* terms to a Boolean constant:

```
def eval(e: Expr[~Var]): Bool =
  choose e {
    // Var case omitted: We can only evaluate closed terms.
    case Cst(b)      => b
    case Not(x)      => not eval(x)
    case Or(x, y)    => eval(x) or eval(y)
    case And(x, y)   => eval(x) and eval(y)
    case Xor(x, y)   => eval(x) != eval(y)
  }
```

The evaluator itself is straightforward. We simply pattern-match on each case and implement the semantics directly. What is interesting is that we cannot evaluate an open term (i.e. a formula with variables in it), hence we simply omit the `Var` case from the pattern-match. The type system then infers that the `eval` function can be passed any Boolean expression as long as it does not use the `Var` variant. This is captured by the type `Expr[~Var]` which is equivalent to `Expr[~{Cst, Not, Or, And, Xor}]`.

► **Example 3 (Variant Elimination).** We can also write a Boolean formula simplifier which eliminates the `Xor` term by translation:

```
def simplify(e: Expr[s]): Expr[~Xor] =
  choose e {
    case Var(x)      => Var(x)
    case Cst(b)      => Cst(b)
    case Not(x)      => Not(simplify(x))
    case Or(x, y)    => Or(simplify(x), simplify(y))
    case And(x, y)   => And(simplify(x), simplify(y))
    case Xor(x, y)   =>
      let x1 = simplify(x);
      let y1 = simplify(y);
      Or(And(x1, Not(y1)), And(Not(x1), y1))
  }
```

The simplifier is also straightforward. The return type of the `simplify` function now excludes the possibility that the returned value can contain a `Xor` variant. This is captured by the type `Expr[~Xor]` which is equivalent to `Expr[~{Var, Cst, Not, Or, And}]`.

The unfortunate weakness of the simplifier is that if we know that the input cannot contain any variables (e.g. the `Var` variant) then this information is lost in the output. For example, if we have a closed Boolean formula, *we cannot simplify it and then evaluate it* because the return type of `simplify` includes the `Var` variant in its type. We lost the knowledge that the term was closed!

17:6 Restrictable Variants

The fundamental issue is that in `simplify` we have lost the relation between the type-level index in the argument type (i.e., `Expr[s]`) and the result type (i.e., `Expr[~Xor]`). To overcome this, we introduce the `choose*` construct. The `choose*` construct allows us to maintain a relation between the input type and the output type, as the following example shows:

► **Example 4 (Structure-Preserving Map)**. We can use the `choose*` construct to write a structure-preserving map function:

```
def map(f: Int32 -> Int32, e: Expr[s]): Expr[s] =
  choose* e {
    case Var(x)      => Var(f(x))
    case Cst(b)      => Cst(b)
    case Not(x)      => Not(map(f, x))
    case Or(x, y)    => Or(map(f, x), map(f, y))
    case And(x, y)   => And(map(f, x), map(f, y))
    case Xor(x, y)   => Xor(map(f, x), map(f, y))
  }
```

The `map` function applies a function $f : \text{Int32} \rightarrow \text{Int32}$ to every variable in the given expression. What is essential is that the argument type is `Expr[s]` and the result type is `Expr[s]` which means that information about the “active” variants in the input is preserved in the output.

► **Example 5 (Simplify – Revisited)**. Recall that the original version of `simplify` used `choose` and had the signature:

```
def simplify(e: Expr[s]): Expr[~Xor] = ...
```

If we change the implementation to use `choose*` we instead get the more precise signature:

```
def simplify(e: Expr[s]): Expr[(s - Xor) + {Not, And, Or}]
```

which captures that `simplify` will return an expression that may contain the `Not`, `Or`, `And` variants plus the `Cst` and `Var` variants, if the input contains them. We might have hoped the return type would simply be `Expr[(s - Xor)]`, but the type system cannot exclude the `Not`, `Or`, `And` variants because they are introduced by elimination of `Xor`. Fortunately, the signature of `simplify` is strong enough to capture the two important properties we care about:

- The `Var` variant can only occur in the output if it occurs in the input.
- The `Xor` variant is eliminated, i.e. cannot occur in the output.

Consequently, if the input is a closed formula (i.e. lacks the `Var` variant) then after simplification *it will still be closed* and we can evaluate it.

With the updated `simplify`, we can write a function:

```
let run = simplify >> eval
```

which is inferred to have the type `Expr[s - Var] → Bool` where the closedness requirement is propagated “backwards” through the (forward) function composition operator `>>`.

► **Example 6 (Substitution)**. We can also write a substitution function that replaces each variable in a Boolean formula with a value from an environment:

```
def subst(m: Map[Int32, Bool], e: Expr[s]): Expr[(s - Var) + Cst] =
  choose* e {
    case Var(x)      => Cst(Map.getWithDefault(x, false, m))
    case Cst(b)      => Cst(b)
    case Not(x)      => Not(subst(m, x))
    case Or(x, y)    => Or(subst(m, x), subst(m, y))
    case And(x, y)   => And(subst(m, x), subst(m, y))
    case Xor(x, y)   => Xor(subst(m, x), subst(m, y))
  }
```

We define the `subst` function to operate on all Boolean expressions. The return type of `subst` is the same as the input type (*sans* `Var`), but may potentially contain the `Cst` variant. The reason is that the type system is not sufficiently expressive to capture that the `Cst` variant can only occur if *either* the `Var` *or* the `Cst` variants are present in the input. This “loss of precision” only affects the `Cst` variant. For example, we still know that if the input cannot contain the `Xor` variant then neither can the output.

► **Example 7 (Function Composition).** Imagine that we have a fast evaluator, but it only supports the `Cst`, `Not`, `And`, and `Or` variants. We can capture this with the signature:

```
def fasteval(e: Expr[s & {Cst, Not, And, Or}]): Bool = ...
```

We can *compose* the `simplify`, `subst`, and `fasteval` functions as follows:

```
let fastrun = m -> simplify >> subst(m) >> fasteval
```

The (inferred) type of `fastrun` is:

$$\text{fastrun} : \forall s. \text{Map}[\text{Int32}, \text{Bool}] \rightarrow \text{Expr}[s] \rightarrow \text{Bool}$$

i.e., given an environment and a Boolean expression it computes a primitive `Bool`.

What is essential is that the function types of `simplify`, `subst`, and `fasteval` *compose* in a way that preserves the information that `simplify` eliminates the `Xor` variant and `subst` eliminates the `Var` variant, hence the final call to `fasteval` is valid. Looking at the types:

$$\begin{aligned} \text{simplify} &: \forall s_1. \text{Expr}[s_1] \rightarrow \text{Expr}[s_1 - \text{Xor} + \{\text{Not}, \text{And}, \text{Or}\}] \\ \text{subst}(m) &: \forall s_2. \text{Expr}[s_2] \rightarrow \text{Expr}[(s_2 - \text{Var}) + \text{Cst}] \\ \text{fasteval} &: \forall s_3. \text{Expr}[s_3 \& \{\text{Cst}, \text{Not}, \text{And}, \text{Or}\}] \rightarrow \text{Bool} \end{aligned}$$

We see that when we apply the output of `simplify` as the input to `subst(m)`, we get the type:

$$\text{Expr}[(((s_1 - \text{Xor} + \{\text{Not}, \text{And}, \text{Or}\})) - \text{Var}) + \text{Cst}]$$

This type is compatible with the input type of `fasteval` because the *set equation*:

$$(((s_1 - \text{Xor} + \{\text{Not}, \text{And}, \text{Or}\})) - \text{Var}) + \text{Cst} = s_3 \& \{\text{Cst}, \text{Not}, \text{And}, \text{Or}\}$$

has a solution. Specifically, it has the most-general unifier:

$$\{s_3 \mapsto s_1 + \{\text{Cst}, \text{Not}, \text{And}, \text{Or}\}\}$$

where s_1 and s_2 are implicitly mapped to themselves. This solution can be found by Boolean unification. Thus, in summary, we are able to infer that `fastrun` has the type $\forall s. \text{Map}[\text{Int32}, \text{Bool}] \rightarrow \text{Expr}[s] \rightarrow \text{Bool}$ which means that it works for any Boolean formula.

As we shall discuss in Section 5, the power of our system is this ability to track the *introduction* and *elimination* of variants *through function composition*. Notably, several other existing systems lack this property, including row-based extensible variants. The key issue is that a row polymorphic system is unable to precisely relate the input type of a (partial) pattern-match to its output type. Thus we lose track of the fact that `simplify` eliminates the `Xor` variant and hence we cannot call `fasteval`. We call this phenomenon the “co-domain” problem for extensible variants since these type systems lack the ability to relate the domain of a (partial) pattern-match (i.e. its input type) to its co-domain (i.e. its output type).

In our experience and based on the case studies (Section 6), we find it important to stress how important this property is for reusability. In a compiler, we want to write the `subst` function *once* and for the *entire* abstract syntax tree. However, if the `subst` function loses information about what variants can be returned in its output, then its utility is hampered, as most compiler phases only operate on a subset of the entire abstract syntax tree.

2.1 Summary

We conclude with a summary of the properties of the proposed system:

- **(Property I)** Restrictable variants are sum types indexed by a type-level set formula that over-approximates the “active” set of labels of the sum. Programmers can use restrictable variants to write *one* data type definition that is reusable in many different contexts.
- **(Property II)** The `choose` construct enables programmers to write non-exhaustive pattern-matches on restrictable variants handling only the relevant cases. The `choose*` construct enables a form of refinement typing where the result type of a non-exhaustive pattern-match is related to its input type.
- **(Property III)** Functions on restrictable variants *compose* under introduction and elimination of labels; i.e., a sequence of introductions and eliminations does not lose information at the type level.
- **(Property IV)** The type system ensures that the non-exhaustive `choose` or `choose*` constructs cannot get stuck at runtime. The type system extends Hindley-Milner and supports complete type inference.
- **(Property V)** Restrictable variants are a natural generalization of algebraic data types and are simple to implement.

3 Restrictable Variants

We now present $\lambda_{\text{var}}^{\text{res}}$: a minimal lambda calculus with restrictable variants. We present its syntax and semantics, then its type system, and finally its meta theoretic properties. The $\lambda_{\text{var}}^{\text{res}}$ calculus and its type system are mostly standard; the novelties are the `choose` and `choose*` constructs and the use of *set formulas* in the type system.

3.1 Syntax and Semantics

We begin with a discussion of the syntax and semantics of the $\lambda_{\text{var}}^{\text{res}}$ calculus.

Syntax

The syntax of the $\lambda_{\text{var}}^{\text{res}}$ calculus (cf. Figure 1a) includes the standard lambda calculus constructs: variables, constants, lambda abstractions, and function applications. The `let`-expression allows polymorphic generalization, as is standard in Hindley-Milner. We include the `if-then-else` expression to illustrate how the type system merges information. We require that every $\lambda_{\text{var}}^{\text{res}}$ program comes with a map $\Sigma : \text{Enum} \rightarrow \text{Label} \rightarrow \text{Scheme}$ of declared variants.

The raison d’être is the `choose $e \{\bar{\eta}\}$` and `choose* $e \{\bar{\eta}\}$` expressions. In both expressions, e is the match expression and $\bar{\eta}$ is a sequence of match cases. A match case is of the form `case $\mathcal{E}.\ell(x) \Rightarrow e$` where \mathcal{E} is the enum that the label ℓ belongs to, x is the match variable, and e is the match expression body. As shown, we prefix all labels with the enum they come from; i.e., we write “Color.Red” and not just “Red”. Recall that both `choose` expressions are needed, since `choose` allows expression bodies to have an arbitrary type, whereas `choose*` requires that the expression bodies have the same type as the match expression (modulo the type-level

$ \begin{aligned} v \in Val &= () \mid \mathbf{true} \mid \mathbf{false} \\ &\mid \lambda x. e \\ &\mid \mathcal{E}.\ell(v) \\ e \in Exp &= x \mid v \mid ee \mid \mathcal{E}.\ell(e) \\ &\mid \mathbf{let} \ x = e \ \mathbf{in} \ e \\ &\mid \mathbf{if} \ e \ \mathbf{then} \ e \ \mathbf{else} \ e \\ &\mid \mathbf{choose} \ e \ \{\bar{\eta}\} \\ &\mid \mathbf{choose}^* e \ \{\bar{\eta}\} \\ &\mid \mathbf{open} \ e \\ \eta \in Case &= \mathbf{case} \ \mathcal{E}.\ell(x) \Rightarrow e \\ \mathcal{E} \in Enum &= \text{a set of enums} \\ \ell \in Tag &= \text{a set of tags} \\ x, y \in Var &= \text{a set of variables} \end{aligned} $	$ \begin{aligned} \varphi \in Formula &= \emptyset \mid \{\mathcal{E}.\ell\} \mid \beta \mid \varphi^b \mid \varphi \cup \varphi \mid \varphi \cap \varphi \\ \tau \in Type &= \alpha \mid \mathbf{Unit} \mid \mathbf{Bool} \mid \tau \rightarrow \tau \mid \mathcal{E}[\varphi] \\ \sigma \in Scheme &= \tau \mid \forall \alpha. \sigma \mid \forall \beta. \sigma \\ \alpha \in TypeVar &= \text{a set of type variables} \\ \beta \in BoolVar &= \text{a set of Boolean variables} \end{aligned} $
---	--

(a) Syntax of $\lambda_{\text{var}}^{\text{res}}$.(b) Types and Type Schemes of $\lambda_{\text{var}}^{\text{res}}$.■ **Figure 1** Syntax and Types of $\lambda_{\text{var}}^{\text{res}}$.

$ \begin{aligned} &\frac{(\lambda x. e) v \rightsquigarrow e[x \mapsto v]}{\mathbf{let} \ x = v \ \mathbf{in} \ e \rightsquigarrow e[x \mapsto v]} \quad (\text{E-APP}) \\ &\frac{}{\mathbf{if} \ \mathbf{true} \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 \rightsquigarrow e_1} \quad (\text{E-LET}) \\ &\frac{}{\mathbf{if} \ \mathbf{false} \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 \rightsquigarrow e_2} \quad (\text{E-ITE-T}) \\ &\frac{}{\mathbf{if} \ \mathbf{false} \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 \rightsquigarrow e_2} \quad (\text{E-ITE-F}) \\ &\frac{}{\mathbf{open} \ \mathcal{E}.\ell(v) \rightsquigarrow \mathcal{E}.\ell(v)} \quad (\text{E-OPEN}) \\ &\frac{\eta_i = \mathbf{case} \ \mathcal{E}.\ell(x) \Rightarrow e}{\mathbf{choose} \ \mathcal{E}.\ell(v) \ \{\bar{\eta}\} \rightsquigarrow e[x \mapsto v]} \quad (\text{E-CHOOSE}) \\ &\frac{\eta_i = \mathbf{case} \ \mathcal{E}.\ell(x) \Rightarrow e}{\mathbf{choose}^* \ \mathcal{E}.\ell(v) \ \{\bar{\eta}\} \rightsquigarrow \mathbf{open} \ e[x \mapsto v]} \quad (\text{E-CHOOSE-}\star) \end{aligned} $	$ \begin{aligned} &\frac{e_1 \rightsquigarrow e'_1}{e_1 e_2 \rightsquigarrow e'_1 e_2} \quad (\text{C-APP}) \qquad \frac{e \rightsquigarrow e'}{ve \rightsquigarrow ve} \quad (\text{C-APP2}) \\ &\frac{e_1 \rightsquigarrow e'_1}{\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \rightsquigarrow \mathbf{let} \ x = e'_1 \ \mathbf{in} \ e_2} \quad (\text{C-LET}) \\ &\frac{e_1 \rightsquigarrow e'_1}{\mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3 \rightsquigarrow \mathbf{if} \ e'_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3} \quad (\text{C-ITE}) \\ &\frac{e \rightsquigarrow e'}{\mathbf{open} \ e \rightsquigarrow \mathbf{open} \ e'} \quad (\text{C-OPEN}) \\ &\frac{e \rightsquigarrow e'}{\mathbf{choose} \ e \ \{\bar{\eta}\} \rightsquigarrow \mathbf{choose} \ e' \ \{\bar{\eta}\}} \quad (\text{C-CHOOSE}) \\ &\frac{e \rightsquigarrow e'}{\mathbf{choose}^* \ e \ \{\bar{\eta}\} \rightsquigarrow \mathbf{choose}^* \ e' \ \{\bar{\eta}\}} \quad (\text{C-CHOOSE-}\star) \end{aligned} $
--	---

■ **Figure 2** Evaluation Rules of $\lambda_{\text{var}}^{\text{res}}$.

indices). We construct a variant value with the $\mathcal{E}.\ell(e)$ expression, e.g. “Color.Red()” where $()$ is the unit value. The $\mathbf{choose} \ e \ \{\bar{\eta}\}$ and $\mathbf{choose}^* e \ \{\bar{\eta}\}$ expressions are akin to pattern-matches, except there are no wildcards, tuple patterns, or nested patterns. Importantly, the **choose** and **choose*** expressions do not have to be exhaustive.

The $\mathbf{open} \ e$ expression is not part of the surface syntax, and is present only during evaluation. Semantically, $\mathbf{open} \ e$ is equivalent to e . Its purpose is explained in Section 3.2.

Semantics

The semantics of $\lambda_{\text{var}}^{\text{res}}$ is a call-by-value operational semantics for the lambda calculus. Figure 2 shows the evaluation rules of $\lambda_{\text{var}}^{\text{res}}$ which are standard except for (E-OPEN), (E-CHOOSE), and (E-CHOOSE- \star). We write $e[x \mapsto v]$ for the capture avoiding substitution of $x \mapsto v$ into e . The congruence rules, prefixed with C, enforce a left-to-right evaluation order. The (E-OPEN) rule reduces a tagged value $\mathcal{E}.\ell(v)$ to itself. The (E-CHOOSE) evaluation rule captures that if we evaluate a tagged value $\mathcal{E}.\ell(v)$ for some value v then we look for a case $\mathbf{case} \ \mathcal{E}.\ell(x) \Rightarrow e$ in the pattern-match. If found, we evaluate the case body, i.e. we step to $e[x \mapsto v]$. The (E-CHOOSE- \star) is very similar, but it instead steps to $\mathbf{open} \ e[x \mapsto v]$.

How $\lambda_{\text{var}}^{\text{res}}$ Programs “Get Stuck”

We briefly illustrate how $\lambda_{\text{var}}^{\text{res}}$ programs may get stuck during evaluation. The obvious reason is when `true` or `false` is applied as a function, or when a lambda expression is used as a condition in an if-then-else. The more interesting case is when a `choose` or `choose*` expression is applied to a variant for which there is no case:

```
choose Green {
  case Red => true
  case Blue => false
}
```

The type system will reject such programs.

3.2 Type System

We now describe the type system of $\lambda_{\text{var}}^{\text{res}}$: its types, type rules, and meta-theory.

Mono Types and Poly Types (Type Schemes)

The types of $\lambda_{\text{var}}^{\text{res}}$ are separated into mono types (τ) and type schemes (σ). The mono types include type variables α , the base types `Unit` and `Bool`, function types $\tau \rightarrow \tau$, and variant types $\mathcal{E}[\varphi]$ which consist of an enum symbol \mathcal{E} indexed by a type-level Boolean set formula φ . The language of formulas, for a given variant type \mathcal{E} , consists of the empty set \emptyset , a singleton set with one label $\{\mathcal{E}.l\}$, Boolean variables β , the complement of a formula φ^c , the union of two formulas $\varphi \cup \varphi$, and the intersection of two formulas $\varphi \cap \varphi$. We write $A - B$ for set difference which is equivalent to $A \cap B^c$. We also write $A <: B$ as an alias for the constraint $A - B = \emptyset$ (i.e. $A <: B \Leftrightarrow A \cap B^c = \emptyset$). Note that the complement of a set is well-defined, since a variant type is declared to have a fixed finite set of labels (which forms the universe).

In principle, to ensure that it always clear what the universe of labels is, we should always index each set formula with its associated variant type \mathcal{E} symbol, e.g. we should write $\varphi_{\mathcal{E}}$. However, we typically omit the enum name when it is clear from the context.

We write $\text{ftv}(\varphi)$ for the variables in φ . A *valuation* ν for a formula φ is an assignment of concrete sets of labels to all of the variables in $\text{ftv}(\varphi)$. In this way, we can view a set formula as a function from concrete sets to a concrete set. Two set formulas φ_1 and φ_2 are *equivalent* (written $\varphi_1 \equiv_{\mathbb{B}} \varphi_2$) if they describe the same function. That is, if $\forall \nu. \nu(\varphi_1) = \nu(\varphi_2)$ where ν must be a valuation of both φ_1 and φ_2 .

Type schemes σ extend types by quantification over type variables α and Boolean variables β . That is, a type scheme is of the form $\forall \bar{\gamma}. \tau$, where $\bar{\gamma}$ is a vector of type variables and Boolean variables. Figure 1b shows the types and type schemes of $\lambda_{\text{var}}^{\text{res}}$.

We define type equivalence as the smallest relation $\equiv_{\mathbb{B}}$ ³ such that:

- $\tau \equiv_{\mathbb{B}} \tau$.
- If $\tau_1 \equiv_{\mathbb{B}} \tau'_1$ and $\tau_2 \equiv_{\mathbb{B}} \tau'_2$ then $\tau_1 \rightarrow \tau_2 \equiv_{\mathbb{B}} \tau'_1 \rightarrow \tau'_2$.
- If $\varphi \equiv_{\mathbb{B}} \varphi'$, then $\mathcal{E}[\varphi] \equiv_{\mathbb{B}} \mathcal{E}[\varphi']$.

For example, we have that $\text{Color}[\{\text{Red}\}^c] \equiv_{\mathbb{B}} \text{Color}[\{\text{Green}, \text{Blue}\}]$. Two types, with set formulas in them, do not have to share the same variables (or even share the same number of variables) to be equivalent. For example: $\text{Color}[\{\text{Green}\}] \equiv_{\mathbb{B}} \text{Color}[(\beta \cap \{\text{Green}\}) \cup \{\text{Green}\}]$.

We define substitutions $S : (\text{TypeVar} \cup \text{BoolVar} \rightarrow \text{Type})$ as assignment of type variables to types and Boolean variables to Boolean formulas. We say the type τ is an instance of type scheme σ , written $\sigma \sqsubseteq \tau$, if $\sigma = \forall \bar{\gamma}. \tau'$ and there exists a type substitution S such

³ We overload the $\equiv_{\mathbb{B}}$ symbol to stand for both Boolean equivalence and type equivalence.

that $\text{dom}(S) = \bar{\gamma}$ and $S(\tau') = \tau$. Moreover, we define a context Γ as a map of bindings $x : \sigma$, and $\text{ftv}(\sigma)$ to be the type variables that occur free in σ , and $\text{ftv}(\Gamma)$ as the union of all free type variables in its range. We also define the generalization of a type $\text{gen}(\Gamma, \tau)$ as $\forall \alpha_1, \dots, \forall \alpha_n. \forall \beta_1, \dots, \forall \beta_n. \tau$ where $\{\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_n\} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma)$.

Variant Declarations

As stated earlier, we require every restrictable variant to be declared. Specifically, we assume that there is a set of enum symbols $Enum$ and map $\Sigma : Enum \rightarrow Label \rightarrow Scheme^4$ which assigns a type scheme to every constructor (label) of the type. We require that the type schemes are of one of the two following forms⁵:

1. $\Sigma(\mathcal{E}.l) = \forall \beta. \tau \rightarrow \mathcal{E}[\beta]$ $\text{ftv}(\tau) = \emptyset$
2. $\Sigma(\mathcal{E}.l) = \forall \beta. \mathcal{E}[\beta] \rightarrow \mathcal{E}[\beta]$

These requirements ensure that:

- A constructor is applied to a simple type (e.g. $\Sigma(\text{Color.Red}) = \forall \beta. \text{Unit} \rightarrow \text{Color}[\beta]$), or
- A constructor is applied to the same variant type, but with the *same* type-level index (e.g. $\Sigma(\text{Expr.Not}) = \forall \beta. \text{Expr}[\beta] \rightarrow \text{Expr}[\beta]$).
- The type scheme of a constructor is always polymorphic function type over β whose result type is of the form $\mathcal{E}[\beta]$.

And that the following lemma holds:

► **Lemma 8** (LABEL-INSTANTIATION). *If two instantiations of the same label type scheme share the same result type then they must share the same argument type.*

$$\Sigma(\mathcal{E}.l) \sqsubseteq \tau_1 \rightarrow \mathcal{E}[\varphi] \quad \wedge \quad \Sigma(\mathcal{E}.l) \sqsubseteq \tau_2 \rightarrow \mathcal{E}[\varphi] \quad \Longrightarrow \quad \tau_1 = \tau_2$$

Intuitively, the result type of an instantiated label type scheme uniquely determines its argument type. The idea is that if we know that $\text{Not}(e) : \text{Expr}[\{\text{Cst}, \text{Not}\}]$ then know that the type of e is also $\text{Expr}[\{\text{Cst}, \text{Not}\}]$. In other words, the type-level index of a restrictable variant also applies to its constituents. This fact is used to show preservation.

Type Rules

Figure 3 shows the declarative type rules of $\lambda_{\text{var}}^{\text{res}}$. A declarative typing judgment is of the form $\Gamma \vdash e : \tau$. As is standard, the context $\Gamma : Var \leftrightarrow Scheme$ is a partial function from variables to type schemes. Most of the type rules are standard.

The (T-EQ) rule states that if an expression e can be typed as τ_1 , that type can be replaced by any equivalent type $\tau_2 \equiv_{\mathbb{B}} \tau_1$. The (T-VAR) rule is the standard Hindley-Milner instantiation rule. It states that if the assumption $x : \sigma$ is in the context, then we can *instantiate* σ to a specific type τ , and conclude $x : \tau$. The (T-LET) rule is the standard Hindley-Milner generalization rule. The rule states that if we can type e_1 as τ_1 under the environment Γ then we may *generalize* the type τ_1 to a type scheme σ , and type e_2 under an extended environment with $x : \sigma$.

The (T-TAG) rule states that we can type a tag expression $\mathcal{E}.l(e)$ with the type $\mathcal{E}[\varphi \cup \{\mathcal{E}.l\}]$ where the type-level formula φ is obtained by instantiating the type scheme associated with the label $\mathcal{E}.l$ to $\tau \rightarrow \mathcal{E}[\varphi \cup \{\mathcal{E}.l\}]$. The reason that the label is not part of the scheme is that we do not want to assume the occurrence of the label when the scheme is used in (T-CHOOSE) and (T-CHOOSE- \star).

⁴ In the implementation the Σ map is simply constructed from the `enum` declarations in the program.

⁵ The calculus does not have tuples, but the extension to tuples and polymorphic enums is straightforward. They are supported in the implementation.

17:12 Restrictable Variants

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \quad (\text{T-ABS})$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \equiv_{\mathbb{B}} \tau_2}{\Gamma \vdash e : \tau_2} \quad (\text{T-EQ})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \quad (\text{T-APP})$$

$$\frac{}{\Gamma \vdash () : \text{Unit}} \quad (\text{T-UNIT})$$

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \quad (\text{T-TRUE})$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}} \quad (\text{T-FALSE})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \text{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad (\text{T-LET})$$

$$\frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau} \quad (\text{T-VAR})$$

$$\frac{\Gamma \vdash e : \mathcal{E}[\varphi]}{\Gamma \vdash \text{open } e : \mathcal{E}[\varphi \cup \varphi'] } \quad (\text{T-OPEN})$$

$$\frac{\Gamma \vdash e : \tau \quad \Sigma(\mathcal{E}.l) \sqsubseteq \tau \rightarrow \mathcal{E}[\varphi \cup \{\mathcal{E}.l\}]}{\Gamma \vdash \mathcal{E}.l(e) : \mathcal{E}[\varphi \cup \{\mathcal{E}.l\}]} \quad (\text{T-TAG})$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (\text{T-ITE})$$

$$\frac{\eta_i = \text{case } \mathcal{E}.l_i(x_i) \Rightarrow e_i \quad \Gamma, x_i : \tau_i \vdash e_i : \tau_{\text{out}} \quad \Gamma \vdash e : \mathcal{E}[\varphi_{\text{in}}] \quad \varphi_{\text{in}} <: \bigcup_i \{\mathcal{E}.l_i\} \quad \Sigma(\mathcal{E}.l_i) \sqsubseteq \tau_i \rightarrow \mathcal{E}[\varphi_{\text{in}}]}{\Gamma \vdash \text{choose } e \{ \bar{\eta} \} : \tau_{\text{out}}} \quad (\text{T-CHOOSE})$$

$$\frac{\eta_i = \text{case } \mathcal{E}.l_i(x_i) \Rightarrow e_i \quad \Gamma, x_i : \tau_i \vdash e_i : \mathcal{E}[\varphi_i^{\text{out}}] \quad \Gamma \vdash e : \mathcal{E}[\varphi_{\text{in}}] \quad \Sigma(\mathcal{E}.l_i) \sqsubseteq \tau_i \rightarrow \mathcal{E}[\varphi_{\text{in}}] \quad \varphi_{\text{in}} <: \bigcup_i \{\mathcal{E}.l_i\} \quad (\varphi_{\text{in}} \cap (\bigcup_i (\varphi_i^{\text{out}} \cap \{\mathcal{E}.l_i\}))) \cup \bigcup_i (\varphi_i^{\text{out}} - \{\mathcal{E}.l_i\}) <: \varphi_{\text{out}}}{\Gamma \vdash \text{choose}^* e \{ \bar{\eta} \} : \mathcal{E}[\varphi_{\text{out}}]} \quad (\text{T-CHOOSE-}\star)$$

$$\text{gen}(\Gamma, \tau) = \forall \bar{\alpha}. \tau \text{ where } \bar{\alpha} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma)$$

■ **Figure 3** Type Rules for $\lambda_{\text{var}}^{\text{res}}$.

The (T-OPEN) rule allows tagged values to be typed with additional labels. Essentially, the (T-OPEN) rule enables a form of weakening, which is necessary for the proof of preservation, without having to introduce general sub-typing into the system, since that would break type inference (recall that the surface language does not have `open e` expressions).

The (T-CHOOSE) rule states that a `choose` expression of the form `choose e { $\bar{\eta}$ }`, where $\eta_i = \text{case } \mathcal{E}.l_i(x_i) \Rightarrow e_i$, can be typed as τ_{out} , if the scrutinee e has the type $\mathcal{E}[\varphi_{\text{in}}]$, each tag's type scheme can be instantiated to $\tau_i^{\text{in}} \rightarrow \mathcal{E}[\varphi_{\text{in}}]$, φ_{in} is less than the union of the handled tags $\mathcal{E}.l_i$, if each result e_i has the type τ_{out} under an environment where x_i has type τ_i . This rule expresses the standard `match` typing conditions, but allows non-exhaustive matches as long as the type of e ensures that the value of e will be handled. In this rule we see why the scheme of labels do not include their tag. If it was included, then φ_{in} would *have* to include the label of the case.

The (T-CHOOSE- \star) is similar to the (T-CHOOSE) rule but with two major differences: First, the type of each result e_i must be of the form $\mathcal{E}[\varphi_i^{\text{out}}]$, and second, a side-condition is posed relating the input and output. The side-condition requires the output φ_{out} to be greater than a union of two set formulas: The first formula represents the set of labels *maintained* in φ_{in} ; i.e. the labels in the type of the cases that matches the label of the case and also exists in φ_{in} . The second formula represents the set of labels *introduced* by each case; i.e. the labels in the type of the cases that does not match the label of the case.

We explain the additional side-condition in the (T-CHOOSE- \star) rule with an example. Assume that we have the program below on the left and we assign the case expressions the types on the right:

<pre> choose* c { case Red => Red case Green => Blue case Blue => Green } </pre>	$\varphi_1^{\text{out}} = \text{Red}$ $\varphi_2^{\text{out}} = \text{Blue}$ $\varphi_3^{\text{out}} = \text{Green}$
--	--

If we instantiate the additional side-constraint (T-CHOOSE- \star), we get:

$$(\varphi_{\text{in}} \cap (\text{R} \cap \text{R}) \cup (\text{G} \cap \text{B}) \cup (\text{B} \cap \text{G})) \cup (\text{R} - \text{R}) \cup (\text{G} - \text{B}) \cup (\text{B} - \text{G}) <: \varphi_{\text{out}} \quad (1)$$

where we have highlighted the two parts of the outer union. This simplifies to:

$$(\varphi_{\text{in}} \cap \text{R}) \cup (\text{G} \cup \text{B}) <: \varphi_{\text{out}} \quad (2)$$

That is, the result may contain Blue and Green, but whether it contains Red is dependent on whether the input contains Red.

If, instead of $\varphi_2^{\text{out}} = \{\text{Blue}\}$, we had assumed $\varphi_2^{\text{out}} = \{\text{Blue}\} \cup \beta$ then we get:

$$(\varphi_{\text{in}} \cap (\text{Red} \cup (\text{Green} \cap \beta))) \cup (\text{Green} \cup \text{Blue}) \cup (\beta - \text{Green}) <: \varphi_{\text{out}} \quad (3)$$

This is sensible because if we later learn that $\beta = \text{Yellow}$ (i.e. the second case could return Blue or Yellow) then the above type reduces to:

$$(\varphi_{\text{in}} \cap \text{Red}) \cup (\text{Green} \cup \text{Blue} \cup \text{Yellow}) <: \varphi_{\text{out}} \quad (4)$$

3.3 Uninhabited Types

The type system of $\lambda_{\text{var}}^{\text{res}}$ admits programs that have uninhabited types. For example:

```

def f(c) =
  choose c { case Red   => ... }; // c must have type Color[s]
  choose c { case Green => ... } // where s <: {Red}

```

Here, the two pattern-matches give rise to the constraints $s <: \{\text{Red}\}$ and $s <: \{\text{Green}\}$. Thus, the type of the formal parameter c is $\text{Color}\{\{\}\}$ which is uninhabited. However, this is not a problem; it simply means we cannot call f .

3.4 Meta Theory

The meta theory for the type system is fairly straightforward. We want to ensure that programs which use `choose` and `choose*` cannot get stuck. In other words, we want to prove the standard progress and preservation theorems.

We begin with the canonical forms lemma extended with typing inversion. The lemma shows that the index of a tagged value over-approximates its label:

► **Lemma 9** (CANONICAL-TAG). *If a value is typed with an enum type then the value must be a label of that enum and the enum index includes the label of the value.*

If $\vdash v : \mathcal{E}[\varphi]$ then for some ℓ, v', τ_1 , and φ' it holds that:

1. $v = \mathcal{E}.\ell(v')$
2. $\vdash v' : \tau_1$

17:14 Restrictable Variants

3. $\varphi \equiv_{\mathbb{B}} \varphi' \cup \{\mathcal{E}.l\}$
4. $\Sigma(\mathcal{E}.l) \sqsubseteq \tau_1 \rightarrow \mathcal{E}[\varphi' \cup \{\mathcal{E}.l\}]$

Another key lemma shows that the `open e` expression enables a form of subtyping (which is used to prove preservation of `choose*`):

► **Lemma 10** (OPEN-TAG). *If a value can be typed as an enum with some index then it can also be typed with a super set of that index.*

If $\vdash \mathcal{E}.l(v) : \mathcal{E}[\varphi]$ then $\vdash \mathcal{E}.l(v) : \mathcal{E}[\varphi \cup \varphi']$.

► **Theorem 11** (PROGRESS). *For any closed, well-typed expression then either it is a value or it can evaluate to another expression.*

If $\vdash e : \tau$ then $e \in \text{Val}$ or $e \rightsquigarrow e'$.

► **Theorem 12** (PRESERVATION). *If a closed well-typed expression can take a step then the new expression can also be typed with the original type.*

If $\vdash e : \tau$ and $e \rightsquigarrow e'$, then $\vdash e' : \tau$.

The proofs are available in the extended version of the paper.

3.5 Type Inference

We can support type inference for $\lambda_{\text{var}}^{\text{res}}$ with a suitable extension of Algorithm W [8, 28] with Boolean unification on set formulas [27]. We can use the type rules from the declarative type system of Figure 3 to systematically obtain a collection of type inference rules. The declarative system uses a typing judgment of the form $\Gamma \vdash e : \tau$, the type inference system extends this to $\Gamma \vdash e : \tau; S$ where S is a substitution. Here the type environment Γ and the expression e can be seen as the input to the type inference algorithm and τ and S as the output. We omit the actual inference rules, but they mostly concern a lot of administration around the careful use of substitutions and the composition of substitutions. As is standard, equalities in the declarative system become unification queries in the inference system.

We solve unification queries on types in the standard way, but when we reach two Boolean set formulas we use Boolean unification to solve the queries. Specifically, we rely on the Successive Variable Elimination (SVE) algorithm [27]. The most interesting aspect is how we translate set formula constraints, in the declarative type rules, into unification queries. This however – by *design* – turns out to be straightforward. Given the (T-CHOOSE) type rule:

$$\frac{\eta_i = \text{case } \mathcal{E}.l_i(x_i) \Rightarrow e_i \quad \Gamma \vdash e : \mathcal{E}[\varphi_{\text{in}}] \quad \boxed{\varphi_{\text{in}} <: \bigcup_i \{\mathcal{E}.l_i\}}}{\Gamma, x_i : \tau_i \vdash e_i : \tau_{\text{out}} \quad \Sigma(\mathcal{E}.l_i) \sqsubseteq \tau_i \rightarrow \mathcal{E}[\varphi_{\text{in}}]} \quad \text{(T-CHOOSE)} \quad \Gamma \vdash \text{choose } e \{\eta\} : \tau_{\text{out}}$$

The interesting part is to translate what is shown in the gray box. Recall that this is the part of the constraint which ensures that the input is upper-bounded by the labels that occur in the pattern-match. We translate this constraint to the Boolean unification query:

$$\varphi_{\text{in}} \cap \left(\bigcup_i \{\mathcal{E}.l_i\} \right)^c \stackrel{?}{=} \emptyset$$

whose most-general unifier will capture exactly the above property. Similarly, we can translate the additional side-condition in (T-CHOOSE- \star) as a unification problem on set formulas.

At the time of writing, the type inference machinery works (c.f. Section 4), but sometimes the substitutions computed by SVE can be very large. Large substitutions lead to large formulas which leads to slow inference. Fortunately, we have good reason to believe that the situation can be improved. We know from the case studies (c.f. Section 6) that most functions have small types (i.e. small formulas). Hence the challenge is to compute them. We think that this should be possible with a more sophisticated implementation of SVE that exploits Boolean technology, such as BDDs or ZDDs [1, 29].

3.6 Subtyping

The type system does not have explicit support for subtyping, but instead, like row-based systems, relies on parametric polymorphism [16, 37]. For example, the if-then-else expression:

```
if (true) then Red else Blue
```

is typable because we can assign the types:

$$\Gamma \vdash \text{Red} : \text{Color}[\{\text{Red}, \text{Blue}\} \cup s] \quad \text{and} \quad \Gamma \vdash \text{Blue} : \text{Color}[\{\text{Blue}, \text{Red}\} \cup s]$$

for some type variable s . We could probably extend the type system with subtyping, but then we would likely lose principal type inference.

3.7 A Few Practical Aspects

We conclude with a discussion of a few practical issues.

- When should a programmer use `choose` or `choose*`? A programmer should use `choose` when he or she wants to partially pattern-match on a subset of labels, but the result can be of any type. On the other hand, a programmer should reach for `choose*` when he or she wants to partially pattern-match on a restrictable variant and the result is *the same restrictable variant*. In this case, `choose*` is preferable because it is structure-preserving; relating the “input” labels to the “output” labels.
- Would it be possible to have one “universal” type that holds all possible variants? Yes, in the limit one could define a single gigantic restrictable variant with all possible labels and then use that type everywhere in the program. In practice, this would probably be cumbersome and confusing. For example, it would seem pointless to merge the `Color` and `Expr` restrictable variants, even though one could conceptually do so.

4 Implementation

We have implemented the $\lambda_{\text{var}}^{\text{es}}$ calculus as an extension of the Flix programming language.

Flix is a functional, imperative, and logic programming language that supports algebraic data types, pattern matching, higher-order functions, parametric polymorphism, type classes, higher-kinded types, first-class Datalog constraints, channel and process-based concurrency, and has a polymorphic type and effect system [23, 21, 22, 24]. The Flix compiler project, including the standard library and tests, is approximately 230,000 lines of code.

Adding restrictable variants required approximately 2,000 lines of code. Most of the code was straightforward; the most complex components were the implementation of the type inference rules and Boolean unification on set formulas.

Flix, with our extension, is open source, ready for use, and freely available at:

<https://flix.dev/> and <https://github.com/flix/flix/>

5 Expressiveness and Comparison to Other Systems

In this section, we compare the expressiveness of restrictable variants to other nominal and/or structural type systems. We focus on type systems that support complete type inference. We remind the reader that in Section 2 we used restrictable variants to express:

```
def simplify(e: Expr[s]): Expr[(s - Xor) + {Not, And, Or}]
def subst(m: Map[Int32, Bool], e: Expr[s]): Expr[(s - Var) + Cst]
def fasteval(e: Expr[s & {Cst, Not, And, Or}]): Bool
```

which allowed us to use function composition to define:

```
let fastrun = m -> simplify >> subst(m) >> fasteval
```

We now discuss our ability to express this in other systems with extensible variants.

5.1 Row Polymorphism à la Wand, Gaster and Jones, and Leijen

Row polymorphism is a classic solution to extensible records and variants [37]. A row polymorphic type system supports three primitive operations [11, 16] on variants which are *injection*, *embedding*, and *decomposition*:

$$\begin{aligned} \langle \ell = _ \rangle : \forall \alpha, r. \alpha \rightarrow \langle \ell : \alpha \mid r \rangle & \quad (\text{injection}) \\ \langle \ell \mid _ \rangle : \forall \alpha, r. \langle r \rangle \rightarrow \langle \ell : \alpha \mid r \rangle & \quad (\text{embedding}) \\ \langle \ell \in _ ? _ : _ \rangle : \forall \alpha, \beta, r. \langle \ell : \alpha \mid r \rangle \rightarrow (\alpha \rightarrow \beta) \rightarrow (\langle r \rangle \rightarrow \beta) \rightarrow \beta & \quad (\text{decomposition}) \end{aligned}$$

The last operation allows us to implement pattern matching. What is important is that each use of the ternary-like conditional ($\ell \in _ ? _ : _$) peels off a variant. Note that if we fail to match on ℓ then we refine the type to $\langle r \rangle$ which we continue with in the else branch.

Leijen gives the example [16]:

```
showEvent e =
  (key in e) ? (c -> showChar(c)) :
    (e' -> (mouse in e')) ? (p -> showPoint(p)) : error()
```

Here the idea is that `showEvent` pattern-matches on an extensible variant of the type:

$$\langle \text{key} : \text{KeyEvent} \mid \text{mouse} : \text{MouseEvent} \rangle$$

using the decomposition operator. Note that the program type-checks because both functions `showChar` and `showPoint` (and `error`) return a value of the same type, i.e. `Unit`.

In any case, the return type of the entire “pattern-match” is β , which means that the returned values must have the same type (modulo row-equivalence). Looking over our three functions, we see that⁶:

We can express the `eval` and `fasteval` functions which are given the types:

$$\begin{aligned} \text{eval} : \langle \text{Cst} : \cdot \mid \text{Not} : \cdot \mid \text{Or} : \cdot \mid \text{And} : \cdot \mid \text{Xor} : \cdot \rangle \rightarrow \text{Bool} \\ \text{fasteval} : \langle \text{Cst} : \cdot \mid \text{Not} : \cdot \mid \text{Or} : \cdot \mid \text{And} : \cdot \rangle \rightarrow \text{Bool} \end{aligned}$$

⁶ For simplicity, we ignore the fact that the data type is recursive. We just focus on the labels themselves.

Here the rows are closed and the two functions accept any Boolean formula as long as it only has one of the listed variants. In particular, we cannot accidentally call `eval` or `fasteval` with an open Boolean formula that has the `Var` label.

We can also express the `simplify` and `subst` functions which are given the types:

$$\begin{aligned} \text{simplify} &: \langle \text{Var} : \cdot \mid \text{Cst} : \cdot \mid \text{Not} : \cdot \mid \text{Or} : \cdot \mid \text{And} : \cdot \mid \text{Xor} : \cdot \rangle \rightarrow \\ &\quad \langle \text{Var} : \cdot \mid \text{Cst} : \cdot \mid \text{Not} : \cdot \mid \text{Or} : \cdot \mid \text{And} \rangle \\ \text{subst} &: \text{Map}[\text{Int32}, \text{Bool}] \rightarrow \langle \text{Var} : \cdot \mid \text{Cst} : \cdot \mid \text{Not} : \cdot \mid \text{Or} : \cdot \mid \text{And} : \cdot \mid \text{Xor} : \cdot \rangle \rightarrow \\ &\quad \langle \text{Cst} : \cdot \mid \text{Not} : \cdot \mid \text{Or} : \cdot \mid \text{And} \mid \text{Xor} : \cdot \rangle \end{aligned}$$

Each function accepts a Boolean expression as input, with any labels, and returns a Boolean formula without the `Xor` and `Var` labels, respectively. However, their row types *cannot capture how the input is related to the output*. Hence, unlike with restrictable variants, when we compose the two functions we lose information about the output. In particular, the information that `simplify` has eliminated the `Xor` variant is lost. Hence we cannot call `fasteval`.

One might wonder if we could give the `simplify` function the type:

$$\text{simplify} : \langle \text{Xor} : \cdot \mid r \rangle \rightarrow \langle r \rangle$$

since that seems to capture what we want. However, this type judgment would be *unsound* since we could instantiate r to $\langle \text{Cst} : \text{Int32} \rangle$ and clearly the implementation of `simplify` is not exhaustive for that label. Moreover, if we fail to mention e.g. `Var` then we could also instantiate r to $\langle \text{Var} : \text{Banana} \rangle$ which is also not sound. Thus the only sound solution must mention *all* the variants that `simplify` is prepared to accept. Thus we lose the relationship between the input and output.

A challenge with extensible records and variants based on rows is the question of whether extensions adds a new field (or variant) or overrides an existing field (or variant). The literature has proposal several solutions to this problem:

Extensible Records and Variants with Qualified Types

Gaster and Jones present a type system for extensible records and variants extended with qualified types [11, 13]. The idea is that rows capture the structure of the record (or variant) while predicates are used to ensure that rows are not extended with labels that are already present. Thus the Gaster and Jones system ensures that a record (or variant) cannot be extended with a label it already has. For example, record extension is given the type:

$$(l := _ \mid _) : (r \setminus l) \Rightarrow \alpha \rightarrow \text{Rec } r \rightarrow \text{Rec } \{l : \alpha \mid r\}$$

where the predicate $(r \setminus l)$ in the qualified type captures that r must not contain the label l .

Extensible Records with Scoped Labels

Leijen proposes a different approach that embraces the idea of duplicate labels in records and variants [16]. In Leijen's type system, extensible records are allowed to have multiple fields with the same name (and of different type). This means that fields are scoped.

For example, we can have a record r with the type:

$$r : \{y : \text{Bool} \mid x : \text{Int32} \mid y : \text{Int32}\}$$

17:18 Restrictable Variants

This means that r has *two* fields named y ; one of type `Bool` and the other of type `Int32`. We can access the former, the outermost, with the expression $r.y$. To access the latter, the innermost, we must first remove the outermost y field. Thus we have to write $(r - y).y$. The advantage of Leijen’s over Gaster and Jones’s is that it has principal types without the need for qualified types. The disadvantage is its somewhat unnatural semantics.

What does this mean for a variant to be scoped? It roughly means that when we pattern-match (i.e. decompose) on an extensible variant we always see the outermost label. To find an inner label, we must decompose once, and then decompose again. In pseudo-code:

```
match v /* has type: { Label: Int32 | Label: String | ... } */ {
  case Label(n) => n + 123
  case rest => match rest /* has type: { Label: String | ... } */ {
    case Label(s) => String.toUpperCase(s)
  }
}
```

where we peel off one layer of v to expose the inner `Label` of type `String`. As Leijen writes, this is a “curious” feature and it is not so clear whether it is useful in practice.

Abstracting Extensible Data Types à la Morris et al.

Morris and McKinna presents a type system that unifies the previous type systems into one framework based on qualified types and row theories [30]. The framework can be instantiated to model the systems of Gaster and Jones and Leijen among others. Importantly, the framework also supports instantiations with row concatenation.

The framework can also support a form of extensible variants with the key constructs:

$$\begin{array}{ll}
 \lambda x . \ell \triangleright x & : \quad \tau \rightarrow \Sigma(\ell \triangleright \tau) & \text{(CONSTRUCTION)} \\
 \lambda x . x / \ell & : \quad \Sigma(\ell \triangleright \tau) \rightarrow \tau & \text{(EXTRACTION)} \\
 \lambda x . \text{inj } x & : \quad \forall z_1 z_2 . z_1 < z_2 \Rightarrow \Sigma z_1 \rightarrow \Sigma z_2 & \text{(INJECTION)} \\
 \lambda x y . x \nabla y & : \quad \forall z_1 z_2 z_3 \tau . z_1 \odot z_2 \sim z_3 \Rightarrow \\
 & \quad (\Sigma z_1 \rightarrow \tau) \rightarrow (\Sigma z_2 \rightarrow \tau) \rightarrow (\Sigma z_3 \rightarrow \tau) & \text{(MATCH)}
 \end{array}$$

The key idea is the use of qualified types with two predicates: the *containment* predicate $z_1 < z_2$ and the *combination* predicate $z_1 \odot z_2 \sim z_3$. Using these qualified type predicates, we can define the operations:

- (CONSTRUCTION) constructs a singleton variant with the label ℓ and type τ .
- (EXTRACTION) destructs a *singleton* variant and extracts the value of the variant.
- (INJECTION) extends a variant with additional labels. The operation uses the *containment* predicate $z_1 < z_2$. Its meaning is dependent on the specific row theory. For example, using a theory that disallows duplicates, it means that the labels of z_1 must be a subset of the labels of z_2 (with compatible types).
- (MATCH) is a combinator that composes two functions which operate on parts of a variant into a single function that works on the row concatenation of their input types. The operation uses the *combination* predicate $z_1 \odot z_2 \sim z_3$. Its meaning is again dependent on the specific row theory. For example, using a theory that disallows duplicates, it means that z_1 and z_2 must be disjoint sets of labels and z_3 must be their union. Note that the return type τ of the two functions must be same, hence the match construct does not relate its input type to its output type.

We illustrate this loss of precision with the following example:

$$\lambda x . \left(\left(\lambda y . \text{inj } (A \triangleright y/A) \right) \nabla \left(\lambda z . \text{inj } (B \triangleright z/B) \right) \right) \left(\text{inj } x \right) \quad :$$

$$\forall z_1, z_2, \tau_1, \tau_2 . z_1 \triangleleft (A \triangleright \tau_1, B \triangleright \tau_2), (A \triangleright \tau_1, B \triangleright \tau_2) \triangleleft z_2 \Rightarrow \Sigma z_1 \rightarrow \Sigma z_2$$

Informally, the function is simple the identity function on a variant with two labels A and B , i.e. it maps A to A and B to B . Assume – without loss of generality – that we work on a row theory based on Gaster and Jones which does not allow duplicate labels in variants.

The function receives a variant x , which is allowed to be a subset of the variant $\Sigma(A \triangleright \tau_1, B \triangleright \tau_2)$. It is first injected to be typable with the complete variant, then it is matched on in two different functions that either assume that the variant was A or B via their respective extraction (y/A or z/B). Lastly, the variant is constructed again in singleton variants and injected into the full variant type.

The intention of the function is clearly shown in the type; the input must be a subset of a variant with A and B and the output must be a superset of a variant with A and B . While this type is correct, it is unfortunately not as precise as we would have hoped. In particular, since the function is actually the identity we would have liked the type: $\forall z . z \triangleleft (A \triangleright \tau_1, B \triangleright \tau_2) \Rightarrow \Sigma z \rightarrow \Sigma z$.

The type systems of Gaster and Jones, Leijen, and Morris and McKinna do not solve the fundamental “co-domain problem” for extensible variants. Rather they expose difficulties with row-based variants which require additional machinery or unnatural semantics to fix. Restrictable variants do not suffer from such issues because they rely on *set formulas*.

5.2 Occurrence Typing à la Castagna

Castagna et al. present an expressive set-theoretic type system with a type-case expression [7]. The type system supports union, intersection, and negation types. In their system, the `Color` type can be represented as the *union* type of three singleton types:

$$\text{Red} \vee \text{Green} \vee \text{Blue}$$

and we can match on these using the type-case expression:

$$e_1 \in \tau ? e_2 : e_3$$

where control flow enters the e_2 branch if e_1 reduces to a value $v : \tau$, or e_3 if it does not; i.e. $v : \neg\tau$. The type-case expression is their powerful alternative to the *if-then-else/match/choose* expression, allowing an association between each possible type of the input and the respective type of the output. For example, in their system, the `isWarm` function can be expressed as:

$$\lambda x . x \in \text{Red} ? \text{True} : (x \in \text{Blue} ? \text{False} : \text{undefined})$$

which has the type:

$$(\text{Red} \rightarrow \text{True}) \wedge (\neg\text{Blue} \rightarrow \text{False})$$

Note that, in the last case, where $x \notin \text{Red}$ and $x \notin \text{Blue}$ the untypable expression `undefined` is used to indicate an unreachable case. The precision of the typing – essentially encoding the entire pattern-match at the type level – is very expressive and solves the “co-domain problem” we have outlined. However, the types can become very complex and unwieldy, and there is limited support recursive types and recursive functions [7]. For example, the `subst(m)` function would be given the large intersection type:

17:20 Restrictable Variants

$$(\text{Var}(\text{Int32}) \vee \text{Cst}(\text{Bool}) \rightarrow \text{Cst}(\text{Bool})) \wedge (\text{Not}(\text{Expr}) \rightarrow \text{Not}(\text{ClosedExpr})) \wedge (\text{Or}(\text{Expr}, \text{Expr}) \rightarrow \text{Or}(\text{ClosedExpr}, \text{ClosedExpr})) \wedge (\text{And}(\text{Expr}, \text{Expr}) \rightarrow \text{And}(\text{ClosedExpr}, \text{ClosedExpr})) \wedge (\text{Xor}(\text{Expr}, \text{Expr}) \rightarrow \text{Xor}(\text{ClosedExpr}, \text{ClosedExpr}))$$

where we also have to define the `Expr` and `ClosedExpr` data types as two large union types.

While the goal of $\lambda_{\text{var}}^{\text{res}}$ is to capture the introduction and elimination of variants, the occurrence typing system goes far beyond this, capturing a large amount of additional information as it maps variant to variant; the cost of the additional information is borne in the complexity of the types. Furthermore, it is not clear that the occurrence typing system is capable of inferring the type of recursive functions, meaning that in order to capture the same *elimination* and *introduction* properties, the programmer would have to provide the large type annotations themselves.

5.3 Relational Nullable Types à la Madsen et al.

Madsen and van de Pol present a relational nullable type system [25]. The type system captures the nullability (i.e. whether an expression may evaluate to null) of an expression in relation to the nullability of other related expressions. For example, using their type system, one can express a function:

```
let f = (host, port) -> match (host, port) {
  case (Absent, Absent)      => ...
  case (Present(h), Present(p)) => ...
}
```

which captures that *either* both `host` and `port` are `Absent` (i.e., “null”) *or* both `host` and `port` are `Present` (i.e., non-“null”). For example, the following two calls type-check:

```
f(Absent, Absent) // OK
f(Present("www.google.com"), Present(80)) // OK
```

whereas the next two calls are rejected by the type system:

```
f(Absent, Present(80)) // NOT OK
f(Present("www.google.com"), Absent) // NOT OK
```

The relational nullable type system associates every expression with a proper type π and a pair of Boolean formulas (φ, ψ) that over-approximate whether the expression *may* evaluate to `Absent` (i.e., null) and *may* evaluate to `Present` (i.e., non-null) [25]. The two Boolean formulas form a small lattice where: `String ? (F, F)` is an uninhabited type (i.e., a type that is neither null nor non-null), and e.g. `String ? (F, ψ)` is the type of non-null Strings.

Relational nullable types and restrictable variants share some similarities:

- The restrictable variants type system use *one* type-level *set formula* to over-approximate the set of variants of an expression, whereas the relational nullable type system uses *two* type-level *Boolean formulas* to over-approximate the nullability and non-nullability of an expression.
- Both systems extend Hindley-Milner with Boolean unification; their system on Boolean formulas and our system on set formulas.
- We find that the relational nullable types tend to be significantly more complex than restrictable variant types. For example, the function from above is given the type:

$$\forall t_1, t_2, t_3, b_1, b_2, b_3, b_4. (t_1, b_1 \wedge \neg b_3 \wedge \neg b_4, b_3) \rightarrow (t_2, b_2 \wedge \neg b_3 \wedge \neg b_4, b_4) \rightarrow t_3$$

5.4 Summary

We believe that restrictable variants offer a new simple and practical sweet-spot in the design space of “extensible” data types. In terms of expressive power, for the programming patterns we have shown, we identify restrictable variants as laying between row-based type systems and full-blown occurrence typing. Importantly, restrictable variants precisely capture the introduction and elimination of variants which leads to better compositionality than row-based variants.

6 Case Studies

We now report on three small case studies that use restrictable variants. The first is the running example of Boolean formulas. The second is a new data structure that combines the `Option`, `List`, and `NonEmptyList` data types. The third is a theoretical study of how restrictable variants can be combined with extensible records to model abstract syntax trees.

6.1 Case Study: Boolean Expressions

We have seen how we can use restrictable variants to represent Boolean formulas. The key idea is that we can use the same data type represent both simple formulas (made from the `Not`, `And`, `Or` connectives) and more complex formulas (e.g. using the `Xor` connective). We can also represent both open and closed formulas (i.e. formulas with or without `Vars`).

6.2 Case Study: Option, List, and NonEmptyList

The Flix standard library supports the three central functional data types: `Options`, `Lists`, and `Nels` (non-empty lists). The `Option` module offers 75 functions and spans 587 lines of code, the `List` module offers 136 functions and spans 1,398 lines of code, and finally the `Nel` module offers 104 functions and spans 703 lines of code. While this “batteries included” approach is great for programmers, the downside is that the implementations of `Option`, `List`, and `Nel` duplicate a lot of functionality. Given that `Option`, `List`, and `Nel` are really just sequences of different lengths ($0 - 1$ for `Option`, $0 - n$ for `List`, and $1 - n$ `Nel`), one might wonder if they could not be unified into one data type. As it turns out, they can!

We can define *one data type* for sequences of integers⁷:

```
enum Seq[s] {
  case Nil
  case One(Int32)
  case Cons(Int32, Seq[s])
}
```

We can then define `Option`, `List`, and `Nel` as type aliases:

```
type alias Option = Seq[{Nil, One}]
type alias List   = Seq[{Nil, Cons}]
type alias Nel    = Seq[{One, Cons}]
```

⁷ Flix naturally supports polymorphic data types, but for simplicity we focus on integer-valued sequences.

17:22 Restrictable Variants

A slightly more general type would be e.g., `type alias Option[s] = Seq[s & {Nil, One}]`. What's important is that we can define common operations on `Seq` *once* and reuse them for different types of sequences.

For example, we can write a `forall` function:

```
def forall(f: Int32 -> Bool, s: Seq[s]): Bool = choose s { ... }
```

And we can also write a `map` function:

```
def map(f: Int32 -> Int32, s: Seq[s]): Seq[s] = choose* s { ... }
```

Importantly, the `map` function preserves information about what variants can occur in the output based on the input. Thus, if we map over an `Option`, we know that the result is an `Option` and if we map over a `Nel`, we know the result is a `Nel`.

We can also write functions that only work for non-empty lists. For example:

```
def head(s: Seq[s - Nil]): Int32 = choose s { ... }
def last(s: Seq[s - Nil]): Int32 = choose s { ... }
```

More interestingly, we can express a function that appends an element to a sequence:

```
def append(elm: Int32, s: Seq[s]): Seq[One, Cons] = choose* s {
  case Nil          => One(w)
  case One(x)       => Cons(x, One(elm))
  case Cons(x, xs) => Cons(x, append(elm, xs))
}
```

The return type of `append`, which is equivalent to `Nel`, captures that the result lacks the `Nil` variant, hence is non-empty. We can use `append` to write a `reverse` function:

```
def reverse(s: Seq[s]): Seq[(s & {Nil}) + One, Cons] = choose* s {
  case Nil          => Nil
  case One(x)       => One(x)
  case Cons(x, xs) => append(x, reverse(xs))
}
```

The type of the `reverse` function is not as precise as we would like. In particular, if we reverse an `Option` type, we lose the information that the sequence has 0 – 1 elements. However, the type is sufficiently precise to capture that if we reverse a non-empty list then the result is also non-empty.

In summary, in our experience, most aggregation functions such as `head`, `forall`, and `count` can be implemented on the `Seq` data type. We can also implement structure preserving functions such as `map`. Where it gets more difficult is with transformations such as `append`, `reverse`, and `flatMap` which do not always have the types we would want. In such cases, we can sometimes implement 2 – 3 functions (corresponding to one for `Option`, `List`, and `Nel`) and thus still have the desired functionality.

6.3 Case Study: Restrictable Variants, Extensible Records

While we have presented *restrictable variants* as a better alternative to *extensible variants*, we have found that it is natural to combine restrictable variants with *extensible records*. For example, returning to the compiler use case, one can imagine an abstract syntax tree that is transformed and decorated with additional information through several compiler phases. We can use restrictable variants to capture the active labels and extensible records to capture the extra information. For example, we can define an abstract syntax tree:

```
enum Expr[s][r: RecordRow] {
  case Cst({value = Bool | r}),
  case Num({value = Int32 | r}),
  case Var({ident = String | r}),
  case Add({e1 = Expr[s, r], e2 = Expr[s, r] | r}),
  case Ite({e1 = Expr[s, r], e2 = Expr[s, r], e3 = Expr[s, r] | r})
  // ...
}
```

Here the the Expr data type has *two* type-level indices: The s index controls the variant part whereas the r index controls the record part. Assume that we also have a data type:

```
enum Type { case TBool, case TInt }
```

then we can use row extension to capture that type inference decorates the AST:

```
def infer(e: Expr[s][r]): Expr[s, (tpe = Type | r)] = ...
```

At the same time, we can also capture that code generation only works for closure-converted, lambda-lifted, and well-typed ASTs:

```
def codeGen(e: Expr[s - Lam][(tpe = Type | r)]): ByteCode = ...
```

This example illustrates that restrictable variants and extensible records complement each other well. We use the variant index to constrain what cases we are prepared to deal with and we use the record index to constrain what additional information we need.

6.4 Pretty Printing Types with Lower- and Upper Bounds

Programmers might find it difficult to read a type signature like:

```
def reverse(s: Seq[s]): Seq[(s & {Nil}) + {One, Cons}]
```

For this reason, we have experimented with showing lower- and upper-bounds of type-level set formulas. For example, the set formula: $\text{Seq}[(s \ \& \ \{\text{Nil}\}) \ + \ \{\text{One}, \ \text{Cons}\}]$ has the lower-bound: $\{\text{One}, \ \text{Cons}\}$ and the upper-bound: $\{\text{Nil}, \ \text{One}, \ \text{Cons}\}$. This means a **choose** or **choose*** must handle the One and Cons variants and may optionally handle the Nil variant.

7 Related Work

We have already discussed how the expressiveness of restrictable variants compares to several other existing systems. In this section, we aim to provide high-level overview of related work.

Row-based Extensible Records and Variants

Wand originally introduced the concept of row variables for an object-oriented setting [37]. A key challenge in the literature on row-based systems has been how to deal with duplicate labels. A challenge that remains to this day [30]. Gaster and Jones present a type system for extensible records and variants that use qualified types [13] to ensure that rows do not contain duplicate labels [11]. Leijen instead propose a type system that permits duplicate labels and gives a semantics to such “scoped” records and variants [16]. Another major challenge has been the question of *row concatenation* [30]. In this direction, Harper and Pierce presents a record calculus and type system that permits record concatenation [12], but lacks type inference. Morris and McKinnon presents a unified framework for row-polymorphic type systems based on *row theories* [30].

Row-based type systems have been used successfully in many applications other than extensible records and variants. For example, type systems based on rows have been used to track exceptions [31], to track effects in algebraic effect systems [17, 18], to model database queries [19], and to type first-class Datalog program values [22].

We refer to Gaster and Jones for a detailed introduction to row polymorphism [11].

Occurrence Typing, GADTs, Constructor Subtyping, and Relational Nullability

Castagna et al. present an occurrence-based type system [7] which uses set-theoretic types to infer precise function signatures. Applied to variants and pattern-matching, the system can track exactly how a function maps labels among each other. The system is purely structural and based on semantic subtyping, whereas our system includes nominal typing.

Generalized algebraic data types (GADTs) extend algebraic data types with additional expressive power by allowing the type scheme of a constructor to restrict its return type [15, 35]. The canonical example is the ability to write an algebraic data type for arithmetic and Boolean expressions $\text{Expr}[\alpha]$ and an evaluation function $\text{eval} : \text{Expr}[\alpha] \rightarrow \alpha$ where α is a type-level index that determines whether the expression evaluates to a `Bool` or `Int`. A significant body of work has focused on how to recover type inference in the presence of GADTs [14, 32]. We think it would be interesting future work to explore possible connections between restrictable variants and GADTs.

Constructor subtyping is an alternative to extensible and restrictable variants where one inductive type τ_1 is considered a subtype of another inductive type τ_2 if τ_2 has more constructors than τ_1 [3, 26]. In relation to restrictable variants, the idea would be to have multiple data types that share similar constructors and then use subtyping to allow functions to operate on multiple of these types.

Madsen and van de Pol propose a type system with support for *relational nullable types* [25]. While a nullable type system tracks whether an expression may evaluate to null based on its type, *relational nullable type systems* track whether an expression may evaluate to null based on its type and the type of *other* related expressions. As discussed, the Madsen and van de Pol system has some similarities to ours: both systems allow partial (non-exhaustive) pattern-matching and both systems are based on Hindley-Milner extended with Boolean unification. However, their system is purely structural and focuses on nullability, whereas our system combines nominal and structural typing.

Refinement Kinds

Luís and Toninho propose refinement typing at the kind level to enable metaprogramming with records [6]. We believe their system could be adapted to variants and pattern matching: The dependent types in their system precisely track the associations between the input and output types of functions. Refinement kinds, however, do not support type inference.

Boolean Unification

Boole studied Boolean unification in the single-variable case and presented a simplified version of the successive variable elimination algorithm [4]. Later, Löwenheim presented another Boolean unification algorithm [20]. Today, an accessible introduction to Boolean unification is provided by Martin and Nipkow [27]. Additional background information is provided by Baader [2], Boudet et al. [5], Robinson and Voronkov [34].

Boolean unification was first used in a type system by de Vries et al. who used it model uniqueness [9]. Later, Madsen and van de Pol presented a polymorphic type and effect system which used Boolean unification for inference [24].

8 Conclusion

We have presented *restrictable variants* as a simple and practical alternative to extensible variants. A restrictable variant is a sum type indexed by a type-level set formula of its active labels. We have also introduced the `choose` and `choose*` pattern-matching constructs which enable non-exhaustive patterns matches on restrictable variants. Notably, the `choose*` construct allow us to precisely track the *introduction* and *elimination* of variants through function composition.

We have presented a type system for a minimal calculus with restrictable variants. The type system, which based on Hindley-Milner extended with type-level set formulas, ensures that non-exhaustive pattern-matches cannot get stuck. The system supports complete inference via a suitable extension of Algorithm W with Boolean unification on set formulas.

We have implemented restrictable variants as an extension of the Flix programming language and used the implementation for a few case studies. The extension is ready for use, freely available, and open-source.

References

- 1 Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, C-27(06), 1978. doi:10.1109/TC.1978.1675141.
- 2 Franz Baader. On the complexity of Boolean unification. *Information Processing Letters*, 67(4), 1998. doi:10.1016/S0020-0190(98)00106-9.
- 3 Gilles Barthe and Maria João Frade. Constructor subtyping. In *Programming Languages and Systems: 8th European Symposium on Programming (ESOP)*, 1999. doi:10.1007/3-540-49099-X_8.
- 4 George Boole. *The mathematical analysis of logic*. Macmillan, Barclay and Macmillan, 1847.
- 5 Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauss. Unification in Boolean rings and abelian groups. *Journal of Symbolic Computation*, 8(5), 1989. doi:10.1016/S0747-7171(89)80054-9.
- 6 Luís Caires and Bernardo Toninho. Refinement kinds: Type-safe programming with practical type-level computation. *Proc. of the ACM on Programming Languages*, 3(OOPSLA), 2019. doi:10.1145/3360557.


- 7 Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. On type-cases, union elimination, and occurrence typing. *Proc. of the ACM on Programming Languages*, 6(POPL), 2022. doi:10.1145/3462306.
- 8 Luis Damas. *Type assignment in programming languages*. PhD thesis, The University of Edinburgh, 1984.
- 9 Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness typing simplified. In *Implementation and Application of Functional Languages: 19th International Workshop (IFL)*, 2008. doi:10.1007/978-3-540-85373-2_12.
- 10 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI)*, 1991. doi:10.1145/113445.113468.
- 11 Benedict R Gaster and Mark P Jones. A polymorphic type system for extensible records and variants. Technical report, University of Nottingham, 1996.
- 12 Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1991. doi:10.1145/99583.99603.
- 13 Mark P Jones. A theory of qualified types. *Science of Computer Programming*, 22(3), 1994. doi:10.1016/0167-6423(94)00005-0.
- 14 Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical report, University of Pennsylvania, 2004.
- 15 Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005. doi:10.1145/1094811.1094814.
- 16 Daan Leijen. Extensible records with scoped labels. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming (TFP)*, 2005.
- 17 Daan Leijen. Koka: Programming with row polymorphic effect types. In *Proc. 5th Workshop on Mathematically Structured Functional Programming (MSFP)*, 2014. doi:10.4204/EPTCS.153.8.
- 18 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009872.
- 19 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proc. of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2012. doi:10.1145/2103786.2103798.
- 20 Leopold Löwenheim. Über das auflösungsproblem im logischen klassenkalkul. In *Sitzungsberichte der Berliner Mathematischen Gesellschaft* 7, 1908.
- 21 Magnus Madsen. The principles of the Flix programming language. In *Proc. of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2022. doi:10.1145/3563835.3567661.
- 22 Magnus Madsen and Ondřej Lhoták. Fixpoints for the masses: programming with first-class Datalog constraints. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428193.
- 23 Magnus Madsen, Jonathan Lindegaard Starup, and Ondřej Lhoták. Flix: A meta programming language for Datalog. In *Proc. of the 4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0)*, 2022.
- 24 Magnus Madsen and Jaco van de Pol. Polymorphic types and effects with Boolean unification. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428222.
- 25 Magnus Madsen and Jaco van de Pol. Relational nullable types with Boolean unification. *Proc. of the ACM on Programming Languages*, 5(OOPSLA), 2021. doi:10.1145/3485487.

- 26 Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. Zero-cost constructor subtyping. In *Proc. of the 32nd Symposium on Implementation and Application of Functional Languages (IFL)*, 2020. doi:10.1145/3462172.3462194.
- 27 Urusula Martin and Tobias Nipkow. Boolean unification - the story so far. *Journal of Symbolic Computation*, 7(3), 1989. doi:10.1016/S0747-7171(89)80013-6.
- 28 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978. doi:10.1016/0022-0000(78)90014-4.
- 29 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of the 30th Design Automation Conference (DAC)*, 1993. doi:10.1145/157485.164890.
- 30 J Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290325.
- 31 François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2), 2000. doi:10.1145/349214.349230.
- 32 François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006. doi:10.1145/1111037.1111058.
- 33 Didier Rémy. Type inference for records in a natural extension of ML. Technical report, University of Pennsylvania, 1990.
- 34 Alan JA Robinson and Andrei Voronkov. *Handbook of automated reasoning*, volume 1. Elsevier and MIT Press, 2001.
- 35 Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1), 2007. doi:10.1145/1180475.1180476.
- 36 Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. In *Proc. of the 19th ACM SIGPLAN international conference on Functional programming (ICFP)*, 2014. doi:10.1145/2628136.2628161.
- 37 Mitchell Wand. Type inference for simple objects. In *Proc. of the Fourth Annual Symposium on Logic in Computer Science*, 1987.

Programming with Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

Magnus Madsen ✉ 

Department of Computer Science, Aarhus University, Denmark

Jaco van de Pol ✉ 

Department of Computer Science, Aarhus University, Denmark

Abstract

We present purity reflection, a programming language feature that enables higher-order functions to inspect the purity of their function arguments and to vary their behavior based on this information. The upshot is that operations on data structures can selectively use lazy and/or parallel evaluation while ensuring that side effects are never lost or re-ordered. The technique builds on a recent Hindley-Milner style type and effect system based on Boolean unification which supports both effect polymorphism and complete type inference. We illustrate that avoiding the so-called ‘poisoning problem’ is crucial to support purity reflection.

We propose several new data structures that use purity reflection to switch between eager and lazy, sequential and parallel evaluation. We propose a `DelayList`, which is maximally lazy but switches to eager evaluation for impure operations. We also propose a `DelayMap` which is maximally lazy in its values, but also exploits eager and parallel evaluation.

We implement purity reflection as an extension of the Flix programming language. We present a new effect-aware form of monomorphization that eliminates purity reflection at compile-time. And finally, we evaluate the cost of this new monomorphization on compilation time and on code size, and determine that it is minimal.

2012 ACM Subject Classification Theory of computation → Program semantics

Keywords and phrases type and effect systems, purity reflection, lazy evaluation, parallel evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.18

1 Introduction

Programming languages are increasingly multi-paradigm. Kotlin and Scala embrace object-oriented, functional, and imperative programming. JavaScript has a functional core and its ecosystem is increasingly adopting a functional style. Rust, a decidedly imperative language, has a functional flavor with support for algebraic data types, pattern matching, and higher-order functions. C# and Java have adopted lambda expressions and added streams.

Nevertheless, the marriage of paradigms is not always a happy one: laziness and parallelism expose a deep rift between functional and imperative programming. The delayed or parallel evaluation of an impure function may cause its side effects to be lost, to occur out-of-order, or to interfere with each other, leading to potentially disastrous consequences. For these reasons, imperative programming languages tend to use eager and sequential semantics everywhere, thus foregoing the potential benefits of lazy and/or parallel evaluation.

Most mainstream languages, such as Java, Kotlin, and Scala, offer access to a limited form of laziness and parallelism with streams. Yet anarchy reigns: the use of side effects in streams can have unpredictable consequences and nothing prohibits stream operations from having side effects, except for stern warnings in the documentation. Stream pipelines are often described as declarative, but in the presence of side effects, they are anything but that.

We propose to overcome these challenges with a new programming language construct that enables higher-order functions to inspect the purity of their function arguments and to vary their behavior based on this information. For example, `List.map` can vary its behavior



© Magnus Madsen and Jaco van de Pol;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 18; pp. 18:1–18:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

18:2 Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

as follows: when given a *pure* function it lazily maps the function over the list, whereas when given an *impure* function it eagerly maps the function over the list. Thus, `List.map` ensures that side effects are never lost or re-ordered while simultaneously allowing lazy evaluation for pure functions. We say that the `List.map` function is *purity reflective*. Similarly, `Set.count` can vary its behavior as follows: when given a *pure* function it performs the counting in parallel over the set, whereas when given an *impure* function it performs the counting sequentially following the order of the elements in the set. Thus, `Set.count` ensures that side effects do not lead to thread-safety hazards (like deadlocks, race conditions), while still admitting parallel evaluation when given a pure function. Purity reflection empowers programmers, and in particular library authors, to write new data structures that selectively use lazy and/or parallel evaluation under the hood, while semantically appearing to their clients *as-if* always under eager, sequential evaluation.

We argue that purity reflection, as a simple form of effect reflection [22], hits a “sweet spot” for practical programming. The distinction between pure and impure functions is straightforward and understandable by ordinary programmers, while at the same time providing sufficient information to be useful. We want to stress that purity reflection *increases the value of effect systems*. In particular, most use cases of type and effect systems focus on soundness, i.e. the ability to rule out certain erroneous programs. This is of course very desirable, but it does not really add any new expressive power to a programming language. With purity reflection (and effect reflection in general), we empower programmers to write new programs that they could not express before. Thus, “fighting the types and effects” now comes with an additional reward. Purity reflection is enabled by a recent technique to infer *fine-grained*, polymorphic effects automatically [27].

In this paper, we implement purity reflection, from end-to-end, in a production compiler. We use the implementation to retrofit existing and implement new data structures. A key implementation technique is the use of an effect-aware form of monomorphization. In theory, this technique could lead to an exponential blow-up in compilation time and code size, but we experimentally show that this is not the case.

In summary, the contributions of this paper are:

- **(Purity Reflection)** We introduce purity reflection, a new programming language feature that enables higher-order functions to inspect the purity of their function arguments and to vary their behavior based on this information. We argue that purity reflection is a “sweet spot” in the design space of effect reflection.
- **(Data Structures)** We propose several new data structures that use purity reflection to switch between lazy and eager, sequential and parallel evaluation, including the `DelayList` and `DelayMap` data structures.
- **(Compilation)** We discuss two compilation strategies supporting purity reflection: one based on extending the runtime to track purity information in closures and the other based on a new form of effect-aware monomorphization. We implement the latter.
- **(Implementation)** We extend the Flix programming language with purity reflection. We believe Flix is the first large-scale programming language development to support any form of effect reflection.
- **(Evaluation)** We experimentally evaluate the impact of effect-aware monomorphization on compilation time and code size. The results show that the overhead is minimal.

2 Motivation

We motivate our idea with an example. We will use the Flix programming language, but our technique is equally applicable to other ML-style programming languages.

2.1 A Word & Line Count Program

Imagine that we want to write a program that determines if a text contains a specific word. We might start with the program fragment:

```
use List.{flatMap, memberOf};
use String.splitOn;
let lines = haystack |> splitOn("\n");
let words = lines |> flatMap(1 -> splitOn(" ", 1));
memberOf(needle, words)
```

The program works as follows: Given two strings: `haystack` and `needle`, the program splits `haystack` into a list of lines, then it `flatMap`s over each line splitting it into a list of `words`, and finally it computes if `words` contains the string `needle`.

The program works as expected and is written in a natural style: We have two local variables: `lines` and `words` that hold understandable intermediate results. Unfortunately, the program is not very efficient. We construct several intermediate lists and these entire lists are not even needed if the search word `needle` occurs early in the text.

If evaluation of `splitOn` and `flatMap` were lazy, then the program would run fast and not require the construction of these large intermediate lists. Instead, `splitOn` and `flatMap` would build and operate on lazy lists, whose elements would be constructed on-demand when needed by `memberOf`. *But*, since Flix is strict, this is not the case at the moment.

Let us imagine that we later decide to extend the program to also count the number of lines and words in the text, reminiscent of the `wc` command from UNIX. Thus, we change the program to:

```
let lineCount = ref 0;
let wordCount = ref 0;
let lines = haystack |> splitOn("\n");
let words = lines |> flatMap(1 -> {
  lineCount := deref lineCount + 1;
  let ws = splitOn(" ", 1);
  wordCount := deref wordCount + length(ws);
  ws
});
println("Lines: ${deref lineCount}");
println("Words: ${deref wordCount}");
println("Found: ${memberOf(needle, words)}")
```

The extended program is more sophisticated. Running it might print: `Lines: 21, Words: 261, Found: true`. The new program uses a natural style of functional and imperative programming that is common in Java, Kotlin, and Scala. The core of the program remains functional, but the counting is performed in an imperative manner. The program could be written in a purely functional style, but this would require careful threading of state: We would have to operate on triples of the current words on a line and the two counters.

Importantly, this program must be evaluated eagerly. If we were to lazily evaluate `splitOn` and `flatMap` then the two counters would not be updated before they are printed, and the program would print the wrong result (e.g. `Lines: 0, Words: 0, true`).

18:4 Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

The two programs expose a rift between functional and imperative programming. In the functional paradigm we would like certain operations to be lazy to improve performance, whereas in the imperative paradigm it is vital that effectful operations are evaluated eagerly.

We believe that the fundamental tension is between two different views on the essence of operations such as `filter`, `map`, and `flatMap`. In the *imperative view*, these operations *eagerly* transform one data structure into another data structure. If the transformation has any side effects, these occur immediately and in a deterministic order (e.g. the order of a list, the natural order of a tree, etc.). In the *declarative view*, these operations *describe* how a data structure should be transformed, but the transformation is not applied until *needed*. In this view, effectful transformations are evil; either banned outright (like in Haskell) or strongly discouraged with stern warnings (like in Java, Scala).

So what can be done? In this paper, we propose a technique, i.e. purity reflection, where we can have our cake and eat it too. Purity reflection allows both programs – *exactly as written* – to compute their expected results while the first is evaluated lazily and the second is evaluated eagerly. This allows us to *write programs the way we want* while ensuring that side effects are never lost and always occur in the expected order.

We can use purity reflection to switch between eager and lazy evaluation, but our technique is equally applicable to switching between sequential and parallel evaluation. For example, if we know that the predicate function passed to `Set.count` is pure, then it is safe to evaluate the function in parallel over disjoint subsets of the set.

2.2 Streams: An Unsound Solution

Before we proceed, we want to highlight the challenges posed by trying to combine side effects, laziness, and parallelism in a single programming language. Mainstream programming languages, such as Java and Scala, support a small collection of data structures that are lazy and/or parallel. Most prevalent is the support for *streams*, a lazy (and sometimes parallel) data structure that represents a sequence of elements.

2.2.1 Java

The `java.util.Stream` package offers a collection of utilities for working with “sequences of elements supporting sequential and parallel aggregate operations”. The documentation for the package states that¹:

“side effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards.”

A bit later, the documentation goes on to state:

“[...] The ordering of side effects may be surprising. [...] The eliding of side effects may also be surprising. [...]”

In total, the documentation for `Stream` warns about side effects almost twenty times!

¹ <https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html>

2.2.2 Scala

The `scala.collection.parallel` package offers a collection of parallel data structures. The documentation for the package states that²:

[...] These concurrent and “out-of-order” semantics of parallel collections lead to the [...] implications:

- *Side effecting operations can lead to non-determinism*
- *Non-associative operations lead to non-determinism*

Given the concurrent execution semantics of the parallel collections framework, operations performed on a collection which cause side effects should generally be avoided, in order to maintain determinism.”

The documentation for `ParIterable` goes on to state³:

[...] Since implementations of bulk operations may not be sequential, this means that side effects may not be predictable and may produce data-races, deadlocks or invalidation of state if care is not taken. [...]

As these examples illustrate, the combination of effectful operations with lazy and/or parallel evaluation is fraught with danger. A mindful programmer is left weary. Perhaps as a consequence, a study by Khatchadourian et al. finds that “stream parallelization is rarely used”, despite the fact that “streams tend not to have side effects” [17].

This paper provides a path out of the quagmire.

2.3 Proposed Solution

We propose a solution based on the following simple idea:

Data structure operations (such as `map`, `filter`, ...) may use lazy or parallel evaluation when they are given pure function arguments, but revert to eager, sequential evaluation when given impure function arguments to ensure that side effects are not lost and that the order of effects is preserved.

We illustrate this idea with two examples:

```
@LazyWhenPure
def map(f: a -> b & ef, l: List[a])
  = reifyEff(f) {
    case Pure(g) => mapL(g, l)
    case _       => mapE(f, l)
  }

@ParallelWhenPure
def cnt(f: a -> Bool & ef, s: Set[a])
  = reifyEff(f) {
    case Pure(g) => parCnt(g, s)
    case _       => fold(...)
  }
```

The program construct `reifyEff(exp)` allows us to reflect on the purity of the closure `exp`. In the program fragment on the left, which implements the `map` function on a list, we use `reifyEff` to inspect the purity of the function argument `f`. If it is pure, we use `mapL` to lazily apply the function `f` over the list (i.e. no evaluation happens yet). If, on the other hand, `f` is impure we use `mapE` to immediately apply `f` eagerly over the elements of the list.

² <https://docs.scala-lang.org/overviews/parallel-collections/overview.html>

³ <https://www.scala-lang.org/api/2.12.2/scala/collection/parallel/ParIterable.html>

18:6 Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

Note that in the pure case, `reifyEff` rebinds `f` as `g` (i.e. it is the same function), but now `g` is typed as a pure function. This rebinding avoids the need for flow-sensitive typing.

In the program fragment on the right, we use `reifyEff` to determine whether to count the elements that satisfy a given predicate `f` sequentially or in parallel over a set. If `f` is pure then we perform the counting in parallel, otherwise, we perform it sequentially using an ordinary fold.

The annotations `@LazyWhenPure` and `@ParallelWhenPure` have no semantic meaning, but serve as documentation for the programmer.

The `reifyEff` construct is enabled by a recent Hindley-Milner style type and effect system that supports effect polymorphism, type inference, and computes purity information for every sub-expression in a program [27]. Building on this type and effect system, we can implement purity reflection as a compile-time programming construct that is eliminated by a new form of effect-aware monomorphization. For example, we will monomorph two versions of the `map` function: one for pure functions and one for impure functions.

We now discuss some properties of the proposed solution:

- (*Modular*) The technique supports abstraction: A library *author* can implement data structure operations that make selective use of lazy or parallel evaluation without leaking those details to the client. A library *user* can reason about his or her code *as-if* under eager and sequential semantics.
- (*Gradual*) It is easy to start using the technique: A data structure can be made gradually lazy or parallel without affecting the semantics of its clients[†].
- (*Programmable*) The technique is based on a new programming language construct. Thus, maximum power is placed in the hands of library authors (and programmers in general) who may have better knowledge of when to exploit laziness or parallelism.
- (*Zero Cost*[‡]) The new programming construct can be eliminated entirely at compile-time. Thus programs using the technique suffer no runtime overhead.
- (*Sound*^{††}) The technique is based on a sound type and effect system: It ensures that if an expression is pure then it cannot have a side effect. The typing of lazy expressions ensures that side effects cannot be hidden and later revealed.

[†] Of course, programmers and library authors should be aware that (i) switching from eager to lazy evaluation can potentially lead to space leaks, and (ii) switching from sequential to parallel evaluation may slow down the program. However, we believe both situations can be managed. For (i), lazy evaluation should only be used for stream-like data structures where space leaks are less likely to occur, and for (ii), parallel evaluation should use light-weight threads and only be enabled for sufficiently large data structures.

[‡] We use an effect-aware form of monomorphization that specializes (i.e. copies) higher-order functions based on the purity of their function argument(s). This ensures that there is no runtime overhead, but it could potentially lead to increased compilation time and increased code size. In Section 7 we experimentally evaluate this cost.

^{††} The technique does not magically guarantee correctness. For example, a programmer could mistakenly implement `List.map` to always return the empty list when given a pure function argument. This does not violate the soundness of the type and effect system itself, but it does violate the commonly understood specification of what `List.map` should do.

$c \in Cst = () \mid \text{true} \mid \text{false} \mid \dots$ $v \in Val = c \mid \lambda x. e$ $e \in Exp = x \mid v \mid e e$ $\mid \text{let } x = e \text{ in } e$ $\mid \text{if } e \text{ then } e \text{ else } e$ $\mid \text{lazy } e \mid \text{force } e$ $\mid \text{print } e$ $x, y \in Var = \text{a set of variables}$	$\tau \in Type = \alpha \mid \iota \mid \tau \xrightarrow{\varphi} \tau \mid \text{lazy } \tau$ $\varphi \in Formula = \mathbf{T} \mid \mathbf{F} \mid \beta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$ $\sigma \in Scheme = \tau \mid \forall\alpha. \sigma \mid \forall\beta. \sigma$ $\iota \in BaseType = \text{Unit} \mid \text{Bool} \mid \text{Int} \mid \dots$ $\alpha \in TypeVar = \text{a set of type variables}$ $\beta \in BoolVar = \text{a set of Boolean variables}$
---	--

(a) Expressions of $\lambda_{\mathcal{B}}$.(b) Types of $\lambda_{\mathcal{B}}$.

■ **Figure 1** Syntax and Types of $\lambda_{\mathcal{B}}$.

3 Purity Reflection

We begin with a brief introduction to the $\lambda_{\mathcal{B}}$ calculus and its Hindley-Milner-style type and effect system [12, 30, 7]. The system is from [27] but extended with the standard `lazy` and `force` constructs [31]. The $\lambda_{\mathcal{B}}$ calculus is the foundation for the Flix programming language implementation (which we build on top of). The $\lambda_{\mathcal{B}}$ calculus is proven sound in [27]. In Section 3.4, we propose a simple extension that requires just one new expression and one new type rule.

3.1 A Minimal Calculus

Syntax

The syntax of $\lambda_{\mathcal{B}}$ includes the standard lambda calculus constructs: variables, constants, lambda abstractions, and function application. As is standard in Hindley-Milner style type systems, the `let`-expression `let $x = e_1$ in e_2` supports polymorphic generalization of e_1 ⁴. The `if-then-else` expression is standard and included to illustrate how the type and effect system merges information from different control-flow paths. The `print` expression is included to have a side effect in the calculus. We add `lazy e` and `force e` to suspend and resume computations. We assume that `force e` uses memoization. Figure 1a shows the syntax of $\lambda_{\mathcal{B}}$.

Semantics

We assume a standard call-by-value semantics, i.e., function arguments are reduced to values before they are substituted into the body of a lambda abstraction. The same applies to `let`-bindings. The only exceptions are `if-then-else`, which uses short circuiting semantics, and `lazy` expressions, which are treated as closures that are computed only once when forced, and then memoized.

⁴ In Flix, which has mutable references, `let`-generalization is subject to the value restriction [8].

$$\begin{array}{c}
 \frac{\Gamma \vdash e : \tau_1 \& \varphi_1 \quad \tau_1 \equiv \tau_2 \quad \varphi_1 \equiv \varphi_2}{\Gamma \vdash e : \tau_2 \& \varphi_2} \text{ (T-EQ)} \\
 \frac{\text{typeOf}(c) = \sigma \quad \sigma \sqsubseteq \iota}{\Gamma \vdash c : \iota \& \mathbf{T}} \text{ (T-CST)} \\
 \frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau \& \mathbf{T}} \text{ (T-VAR)} \\
 \frac{\Gamma \vdash e : \mathbf{String} \& \varphi}{\Gamma \vdash \text{print } e : \mathbf{Unit} \& \mathbf{F}} \text{ (T-PRT)} \\
 \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \& \varphi}{\Gamma \vdash \lambda x. e : \tau_1 \xrightarrow{\varphi} \tau_2 \& \mathbf{T}} \text{ (T-ABS)} \\
 \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi} \tau_2 \& \varphi_1 \quad \Gamma \vdash e_2 : \tau_1 \& \varphi_2}{\Gamma \vdash e_1 e_2 : \tau_2 \& \varphi_1 \wedge \varphi_2 \wedge \varphi} \text{ (T-APP)} \\
 \frac{\Gamma \vdash e : \tau \& \mathbf{T}}{\Gamma \vdash \text{lazy } e : \text{lazy } \tau \& \mathbf{T}} \text{ (T-LAZY)} \\
 \frac{\Gamma \vdash e : \text{lazy } \tau \& \varphi}{\Gamma \vdash \text{force } e : \tau \& \varphi} \text{ (T-FORCE)} \\
 \frac{\Gamma \vdash e_1 : \tau_1 \& \varphi_1 \quad \Gamma, x : \text{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2 \& \varphi_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \& \varphi_1 \wedge \varphi_2} \text{ (T-LET)} \\
 \frac{\Gamma \vdash e_1 : \mathbf{Bool} \& \varphi_1 \quad \Gamma \vdash e_2 : \tau \& \varphi_2 \quad \Gamma \vdash e_3 : \tau \& \varphi_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \& \varphi_1 \wedge \varphi_2 \wedge \varphi_3} \text{ (T-ITE)} \\
 \text{gen}(\Gamma, \tau) = \forall \alpha_1, \dots, \forall \alpha_n. \forall \beta_1, \dots, \forall \beta_n. \tau \quad \text{where } \{\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_n\} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma)
 \end{array}$$

■ **Figure 2** Type Rules for $\lambda_{\mathcal{B}}$ with judgments of the form $\Gamma \vdash e : \tau \& \varphi$.

3.2 Type and Effect System

Types

The types of $\lambda_{\mathcal{B}}$ are separated into monotypes (τ) and type schemes (σ). The monotypes include type variables α , a set of base types ι , and function types $\tau_1 \xrightarrow{\varphi} \tau_2$ that represents functions from values of type τ_1 to values of type τ_2 with *latent* effect φ . We use the type $\text{lazy } \tau$ to denote suspended computations. The type schemes of $\lambda_{\mathcal{B}}$ include monotypes τ and quantified types $\forall \alpha. \sigma$ and $\forall \beta. \sigma$, where α is a type variable and β is a Boolean effect variable. Figure 1b shows the types and type schemes of $\lambda_{\mathcal{B}}$.

In $\lambda_{\mathcal{B}}$ the language of effects is a single Boolean formula φ , i.e. there is only a single “effect”: impurity. If the Boolean formula is equivalent to **true** (**T**) then the expression it describes must be pure. If the Boolean formula is equivalent to **false** (**F**) then the expression may have a side effect. A Boolean formula with variables in it captures the conditions under which the expression is pure. The system is over-approximating: An expression typed as pure *cannot* have a side effect whereas an expression typed as impure *may* have a side effect [27].

Type Judgements

Figure 2 shows the type rules of $\lambda_{\mathcal{B}}$. We define a context Γ as a *partial function* of bindings $x : \sigma$ from variables to type schemes. We also define $\text{ftv}(\sigma)$ to be the type variables that occur free in σ , and $\text{ftv}(\Gamma)$ as the union of all free type variables in its range. A type judgement is of the form $\Gamma \vdash e : \tau \& \varphi$, which states that under type environment Γ , the expression e has type τ and effect φ , where φ is a Boolean formula that captures when the expression is pure.

We now briefly discuss the most important type rules. Except for (T-EQ), the rules are syntax-directed. The (T-CST) rule states that a constant expression is pure. The (T-ITE) rule states in an *if* e_1 *then* e_2 *else* e_3 the overall effect is $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where φ_i is the effect of expression e_i . The (T-ABS) and (T-APP) rules type lambda abstractions and applications. An abstraction takes the effect φ of an expression e and moves it onto the arrow type whereas an application releases the latent effect of the arrow type. The (T-VAR) and (T-LET) rules are the standard Hindley-Milner type rules. We add the (T-LAZY) and (T-FORCE) rules. Note that only pure expressions can be suspended. Thus effects cannot be delayed. The (T-EQ) rule states that we can use type equivalence. In $\lambda_{\mathcal{B}}$ two types are considered equivalent modulo Boolean equivalence.

For example, the following two functions types are equivalent:

$$\text{Int} \xrightarrow{x \vee \neg x} \text{Int} \quad \equiv_{\mathbb{B}} \quad \text{Int} \xrightarrow{\mathbf{T}} \text{Int}$$

By a suitable extension of Algorithm W with Boolean unification, the type and effect system supports complete type inference. We refer to [27] for the full details.

3.3 Effect Polymorphism

The $\lambda_{\mathcal{B}}$ calculus supports effect polymorphism, i.e. the effect of a higher-order function may depend on the effects of its function arguments. For example, the `List.map` function can be given the type:

$$\text{List.map} : \forall \alpha_1, \alpha_2, \beta. (\alpha_1 \xrightarrow{\beta} \alpha_2) \xrightarrow{\mathbf{T}} \text{List}[\alpha_1] \xrightarrow{\beta} \text{List}[\alpha_2]$$

which can be read as: the effect of `List.map` is the same as the effect of its function argument, i.e. `List.map` is pure if its function argument is.

Forward function composition `»` can be given the type:

$$\text{»} : \forall \alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2. (\alpha_1 \xrightarrow{\beta_1} \alpha_2) \xrightarrow{\mathbf{T}} (\alpha_2 \xrightarrow{\beta_2} \alpha_3) \xrightarrow{\mathbf{T}} (\alpha_1 \xrightarrow{\beta_1 \wedge \beta_2} \alpha_3)$$

which can be read as: the composition of f and g is pure if both are pure. Note that the purity of `»` is constructed from the purity of both f and g .

3.4 Purity Reflection with ReifyEff

We extend the $\lambda_{\mathcal{B}}$ calculus with a single new expression:

$$\text{reifyEff}(e_1)\{\text{case Pure}(f) \Rightarrow e_2, \text{case } _ \Rightarrow e_3\}$$

The idea is that if e_1 evaluates to a pure function v then it is bound to f and the whole `reifyEff` expression reduces to $e_2[f \mapsto v]$. Otherwise, the expression reduces to e_3 . Of course, one cannot in general determine whether a function is pure. Thus, we rely on the type and effect system to tell us whether a function value is pure. In other words, in the extended $\lambda_{\mathcal{B}}$ calculus (and Flix in general), only well-typed terms have an operational semantics [35]. In Section 5 we discuss two compilation strategies for how to implement the `reifyEff` construct.

The type rule for the `reifyEff` expression is straightforward and shown in Figure 3. The type rule requires that the expression e_1 has a function type $\tau_1 \xrightarrow{\varphi} \tau_2$ where φ is the latent effect of the function. We type check e_2 in an extended environment, where we introduce a new binder f for the function which is typed as pure (i.e., with effect \mathbf{T}). We *do not* introduce a new binder for the case where the function is impure. This asymmetry is for two reasons:

- The type and effect system is over-approximating: If an expression is pure then it cannot have a side effect, but the opposite is not true: an impure expression is not guaranteed to produce a side effect.

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi} \tau_2 \ \& \ \varphi_1 \quad \Gamma, f : \tau_1 \xrightarrow{\mathbf{T}} \tau_2 \vdash e_2 : \tau_3 \ \& \ \varphi_2 \quad \Gamma \vdash e_3 : \tau_3 \ \& \ \varphi_3}{\Gamma \vdash \text{reifyEff}(e_1)\{\text{case Pure}(f) \Rightarrow e_2, \text{case } _ \Rightarrow e_3\} : \tau_3 \ \& \ \varphi_1 \ \wedge \ \varphi_2 \ \wedge \ \varphi_3} \quad (\mathbf{T}\text{-REIFY-EFF})$$

■ **Figure 3** Type rule for the `reifyEff` construct.

18:10 Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

- If we had introduced a new binder for f and given it the effect `impure` (i.e., effect `F`) then any use of f inside e_3 would cause the whole expression to be impure. But this would prevent purity reflection. We would not be able to use `reifyEff` inside `List.map` while keeping it effect-polymorphic.

3.4.1 Correctness

The correctness of the approach depends on the soundness of the type and effect system and completeness of type inference [27]. There, it is proved that the type system enjoys the progress- and preservation-property. Also, Algorithm W extended with Boolean unification will always compute the most general type and effect. Finally, it is proved that a pure expression can at no time perform an effectful step. So it is not possible to hide effects. Moreover, the `lazy` construct in Flix can only be applied to pure expressions, so it impossible to delay effects.

Of course, nothing in the type and effect system ensures that an implementation satisfies its specification. For example, a programmer could accidentally reverse a list before mapping an effectful function over it. In that case, the effects will happen in the wrong order. Ensuring *functional correctness*, i.e., that a function respects its specification of return values and emitted effects, is generally beyond the scope of Hindley-Milner style type systems.

3.5 Fine-Grained Purity and the Poisoning Problem

We stress that the type rules are *compositional* and *fine-grained*: the purity of an expression is constructed from the purity of its sub-expressions. This is in contrast to the situation in row-polymorphic type and effect systems [39, 20], where the effects of the sub-expressions are required to be the same. Such systems suffer from the so-called *poisoning problem* [40], where the effect of a sub-expression is over-approximated to fit its context.

We illustrate this issue with an example:

```
airplanes |>
  List.map(plane -> plane.pilot) |>
  List.map(pilot -> pilot.name) |>
  List.foreach(println)
```

Here a row polymorphic system infers that sub-expression `List.foreach(println)` has the `PRINT` effect. This in turn pollutes every sub-expression with the `PRINT` effect. Consequently, the row polymorphic system cannot be used to infer that the first two `List.map` operations are pure (and could be applied lazily with our technique). However, the Boolean effect system can infer that each of the `List.map` operations is given pure arguments, even through the polymorphic usage of the pipeline function `|>`. This example demonstrates that for purity reflection to work, one needs a *compositional* and *fine-grained* type and effect system.

3.6 Purity Reflection: A Sweet Spot

We believe that purity reflection hits a “sweet spot”. First, it is simple to explain to programmers: they only have to understand the distinction between pure and impure functions. Second, it requires us to maintain minimal information to implement, either at runtime or compile-time, as discussed in Section 5. Third, as we argue below, a richer effect system may be difficult to exploit in practice. In particular, it is difficult to determine when two effects may interfere. For example:

- **(Aliasing)** Given two effects $\text{READ}(p_1)$ and $\text{WRITE}(p_2)$ where p_1 and p_2 are pointers to mutable memory, can we safely evaluate them lazily or in parallel? The answer depends on whether p_1 and p_2 are aliased, i.e., can point to the same memory location. If they are, then any re-ordering or parallel execution may change the meaning of the program. Unfortunately, we cannot statically know if p_1 and p_2 are aliases without additional heavy machinery: either alias analysis or a sub-structural type system. To solve this, one needs more information, such as fine-grained regions [15, 37, 10].
- **(External Aliasing)** Given two effects $\text{READFILE}(f_1)$ and $\text{WRITEFILE}(f_2)$ where f_1 and f_2 are file paths, can we safely evaluate them lazily or in parallel? As before, the answer depends on whether f_1 and f_2 refer to the same file. We cannot statically determine if f_1 and f_2 may denote the same filename without some notion of control- and data flow analysis. Worse, even, if f_1 and f_2 are guaranteed to be distinct strings, the two file paths may still refer to the same file due to symbolic links in the underlying file system.
- **(Implicit Dependencies)** Given two effects WRITEFILE and CURRENTTIME , can we safely evaluate them lazily or in parallel? Maybe, but not if the programmer is trying to measure the time it takes for the WRITEFILE operation to complete.

These examples do not imply that the task is impossible. If we had a specification of each effect, i.e., if we had much more information from the programmer, we could probably apply lazy and/or parallel evaluation more aggressively. Instead, our system makes the simple and sound assumption that effects should never be omitted nor re-ordered.

4 Four New Data Structures

We now illustrate how `reifyEff` can be used to extend two existing and implement two new data structures that make selective use of lazy and/or parallel evaluation.

4.1 From List to LazyList to DelayList

4.1.1 From List to LazyList

In Flix, the familiar definition of `List` is:

```
enum List[a] {
  case Nil,
  case Cons(a, List[a])
}
```

A list is either the empty list `Nil` or a cons cell `Cons(x, xs)` with an element `x` and a tail `xs`. We can implement list operations such as `filter`, `map`, and `flatMap` in the standard way.

The definition of `List` does not permit lazy evaluation. We can fix that by redefining `List` to have a lazy tail:

```
enum List[a] {
  case Nil,
  case Cons(a, Lazy[List[a]])
}
```

Flix has two expressions to support lazy evaluation: `lazy e` and `force e`. The former suspends the evaluation of an expression `e` returning a thunk of type `Lazy[t]` where `t` is the type of the expression. The latter evaluates a thunk and memoizes the result. Recall that only pure expressions can be suspended. With the updated definition of `List`, we can express eager and lazy versions of every list operation.

18:12 Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

For example, here is the lazy definition of `map`⁵:

```
def mapL(f: a -> b, l: List[a]): List[a] =
  match l {
    case Nil          => Nil
    case Cons(x, xs) =>
      // The tail is *not* yet evaluated.
      Cons(f(x), lazy mapL(f, force xs))
  }
```

The `mapL` function takes a pure function f from values of type a to type b and a list l of elements of type a . If the list is empty, it returns the empty list. Otherwise, there is a `Cons(x, xs)` cell where the tail xs is lazy. In this case, we evaluate f on the head x and construct a lazy computation that maps f over the rest of the list xs . Thus, no evaluation of the tail happens until it is needed. Note that the use of `lazy` and `force` requires the suspended computation to be pure. Consequently, f must be pure, as reflected in the function signature.

We can also implement an eager and *effect-polymorphic* version of `map`⁶:

```
def mapE(f: a -> b & ef, l: List[a]): List[a] & ef =
  match l {
    case Nil          => Nil
    case Cons(x, xs) =>
      let hd = f(x); // Eagerly evaluate f(x)
      let tl = mapE(f, force xs); // Force the rest of the list
      Cons(hd, lazy tl) // Tail is lazy, but fully evaluated
  }
```

The `mapE` function takes a function f from values of type a to type b with latent effect ef and a list l of elements of type a . It pattern-matches on l . If the list is empty it returns the empty list. Otherwise, there is a `Cons(x, xs)` cell where the tail xs is lazy. We evaluate f on the head x ; then we perform a recursive call on the tail xs (forcing the list). Note that moving the tail-computation to a `let`-binding makes it eager. Finally, we return a `cons`-cell with the new head and tail, where the tail is made lazy (but nevertheless has been fully evaluated). Unlike, `mapL`, the `mapE` function permits side effects, because it materializes those effects immediately.

We now have `mapL` and `mapE` which have lazy and eager semantics, respectively. We can use these two functions to define a *purity reflective* `map` function that varies its behavior depending on the purity of its function argument:

```
def map(f: a -> b & ef, l: List[a]): List[b] & ef =
  reifyEff(f) {
    case Pure(g) => mapL(g, l) // Use lazy evaluation.
    case _      => mapE(f, l) // Use eager evaluation.
  }
```

The implementation is straightforward: The `map` function matches on the purity of f . If f is pure, then we bind it to g (which is typed as pure) and call `mapL` passing g . Otherwise, f may be impure, and we call `mapE`.

⁵ The syntax $a \rightarrow b$ denotes a pure function from a to b .

⁶ The syntax $a \rightarrow b \ \& \ ef$ denotes an effect polymorphic function from values of type a to type b with latent effect ef .

Example I

The Flix program fragment⁷:

```
List.range(1, 1_000_000_000) |>           // Lazy
List.map(x -> {println("a"); x + 1}) |>    // Eager
List.map(x -> {println("b"); x * 2})      // Eager
```

Prints one billion a's followed by one billion b's. This takes a while, but ultimately the program terminates. The a's are printed before the b's preserving the order of effects.

Example II

The following Flix program fragment Prints `Some(4)` and terminates immediately. The two map operations are pure, consequently they are applied lazily and only evaluated for the first element of the list.

```
List.range(1, 1_000_000_000) |>           // Lazy
List.map(x -> x + 1) |>                   // Lazy
List.map(x -> x * 2) |>                   // Lazy
List.head |> println                       // Eager in head.
```

Example III

The Flix program fragment:

```
let count = ref 0;
List.range(1, 1_000_000_000) |>           // Lazy
List.map(x -> x + 1) |>                   // Lazy
List.take(1_000) |>                       // Lazy
List.map(x -> {                             // Eager
  count := deref count + 1; x * 2
});
println(deref count)
```

Prints 1000 and terminates rather quickly. The first map operation is applied lazily, and the subsequent take operation is also applied lazily. The final map operation is applied eagerly, but only to the first 1000 elements.

4.1.2 From LazyList to DelayList

While the previous lazy list data structure permits both eager and lazy evaluation, its representation is inefficient. In particular, the lazy list definition has two issues: (i) each use of `Lazy` introduces a layer of indirection. This indirection requires extra memory, slows down access, and puts additional pressure on the garbage collector, and (ii) each `force` operation is guarded by a lock to ensure that the thunk is evaluated at most once. This can cause lock contention and is antithetical to the idea that immutable data structures can be shared freely and efficiently in a concurrent program. To overcome these issues, we actually use the following definition:

⁷ The `List.range(b, e)` function returns a (suspended) list of integers from `b` until `e`.

18:14 Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

```
enum DelayList[a] {  
  case ENil  
  case ECons(a, DelayList[a])  
  case LCons(a, Lazy[DelayList[a]])  
}
```

The idea is that a fully evaluated list is represented with the `ENil` and `ECons` constructors whereas a list with a lazy tail is represented with the `LCons` constructor. We implement all operations to be maximally lazy. Evaluation occurs for two reasons: (i) when required by a terminal operation (e.g., `count` and `foldLeft`), or (ii) when a non-terminal operation is given an impure function argument (e.g., `filter` and `map`).

We have implemented the `DelayList` data structure. The following pure operations are always lazy: `append`, `drop`, `flatten`, `from`, `intercalate`, `intersperse`, `range`, `repeat`, `replace`, `take`, and `zip`. The operations shown in Fig. 4a are lazy when given pure function arguments and otherwise eager.

We can use the `DelayList` data structure to realize the word count example from Section 2.

4.2 From Set to Parallel Set

We have refactored the Flix Standard Library implementation of the `Set` data structure to use parallel evaluation for all of its aggregation operations (shown in Fig. 4b). Each of these functions use `reifyEff` to inspect the purity of their function argument, and then they dispatch to either a sequential or a parallel function that operates on the underlying Red-Black tree. For example, here is the implementation of `Set.count`:

```
@ParallelWhenPure  
pub def count(f: a -> Bool & ef, s: Set[a]): Int32 & ef =  
  reifyEff(f) {  
    case Pure(g) if useParallelEvaluation(s) =>  
      RedBlackTree.parCount(g, s)  
    case _ =>  
      foldLeft((b, x) -> if (f(x)) b + 1 else b, 0, s)  
  }
```

Here we use purity reflection on f to determine whether it is safe to perform the count in parallel or if we must perform it sequentially (going from left to right). Moreover, we use the function `useParallelEvaluation` to estimate whether it is *worth* to perform the count in

<u>@LazyWhenPure</u>		<u>@ParallelWhenPure</u>
<code>def dropWhile(...)</code>		<code>def count(...)</code>
<code>def filter(...)</code>		<code>def map(...)</code>
<code>def filterMap(...)</code>		<code>def mapWithKey(...)</code>
<code>def flatMap(...)</code>	<u>@ParallelWhenPure</u>	<code>def maximumKeyBy(...)</code>
<code>def map(...)</code>	<code>def count(...)</code>	<code>def maximumValueBy(...)</code>
<code>def mapWithIndex(...)</code>	<code>def maximumBy(...)</code>	<code>def minimumKeyBy(...)</code>
<code>def span(...)</code>	<code>def minimumBy(...)</code>	<code>def minimumValueBy(...)</code>
<code>def takeWhile(...)</code>	<code>def productWith(...)</code>	<code>def productWith(...)</code>
<code>def zipWith(...)</code>	<code>def sumWith(...)</code>	<code>def sumWith(...)</code>

(a) `DelayList`: lazy when pure. (b) `Set`: parallel when pure. (c) `Map`: parallel when pure.

■ **Figure 4** Selective Lazy or Parallel datastructures, depending on purity of function arguments.

parallel. In particular, the `useParallelEvaluation` function relies on some heuristics, including the height of the Red-Black tree, to determine whether we *should* use parallel evaluation, given that we *could*.

The implementation of `RedBlackTree.parCount` is straightforward:

```
@Parallel
def parCount(f: (k, v) -> Bool, t: RedBlackTree[k, v]): Int32 =
  match t {
    case Leaf                => 0
    case DoubleBlackLeaf     => 0
    case Node(_, l, k, v, r) => // left, key, value, right
      par (cl <- parCount(f, l);
          cm <- if (f(k, v)) 1 else 0;
          cr <- parCount(f, r))
        yield cl + cm + cr
  }
```

Here we use the built-in Flix construct `par` to evaluate three expressions in parallel. Thus, the count is performed in parallel on the left subtree, on the key and value, and on the right subtree.

We might worry that spawning too many threads may impose an overhead much larger than the time saved by using parallel evaluation. With OS-level threads, which are expensive, this is likely to be the case. A standard solution to this problem is the use of thread pools and/or a fork-join framework. However, with the imminent arrival of light-weight threads in Java (Project Loom), we hope that such administration will no longer be necessary since `VirtualThreads` are very cheap.

4.3 From Map to Parallel Map

We have also refactored the Flix Standard Library implementation of the `Map` data structure (mapping keys to values), to use parallel evaluation for all the aggregation and transformation operations when given pure function arguments (shown in Fig. 4c). The `map` and `mapWithKey` functions are the most interesting since they allow parallel rebuilding of the map when applying a pure function to all of its values. As before, these functions use `reifyEff` to dispatch to the appropriate operation inside `RedBlackTree`.

For example, here is a simplified version `RedBlackTree.parMapWithKey`:

```
@Parallel
pub def parMapWithKey(f: (k, v1) -> v2, t: RedBlackTree[k, v1]):
  RedBlackTree[k, v2] =
  match t {
    case Leaf                => Leaf
    case DoubleBlackLeaf     => DoubleBlackLeaf
    case Node(c, l, k, v, r) =>
      par (l1 <- parMapWithKey(f, l);
          v1 <- f(k, v);
          r1 <- parMapWithKey(f, r))
        yield Node(c, l1, k, v1, r1)
  }
```

4.4 From Map and ParallelMap to DelayMap

We now turn to perhaps the most interesting new data structure: `DelayMap`, a data structure that uses *both* selective lazy and parallel evaluation. A `DelayMap[k, v]` is a map from strict keys (of type `k`) to lazy values (of type `v`). Pure transformations on the values of a `DelayMap`

18:16 Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

are lazy (e.g., `DelayMap.map`). Pure terminal operations use parallelism (e.g., `Delay.count`). The operations in Fig. 5a use lazy evaluation when given pure function arguments. The `unionWith` operation is especially interesting, as it enables the lazy merge of two maps (if the combine operation is pure). The operations in Fig. 5b use parallel evaluation when given pure function arguments. Consider the program fragment:

```
m1 |> DelayMap.map(x -> x + 1)           // Lazy.
      DelayMap.map(x -> x * 2)           // Lazy.
      DelayMap.count(x -> {println(x); x > 5}) // Parallel + sequential
```

Assume we start with a map `m1` from strings to integers. The first and second map operation transform the values and are applied lazily because they are pure. The last count operation is impure, hence must be applied eagerly. However, before doing so, we force the entire map in parallel (i.e., applying `x -> x + 1` and `x -> x * 2` to every value) and only then the counting is performed, sequentially from left to right. This example shows that we apply pure operations lazily, but once we have performed an impure operation, we force the entire data structure in parallel, and then switch back to sequential evaluation.

4.4.1 Example I

We can use a `DelayMap` to write code that is both natural and efficient. Assume that we have two maps m_1 and m_2 of type `DelayMap[String, Int32]`. Each map records the number of occurrences of a specific word drawn from some documents d_1 and d_2 . We can merge the two maps and then compute the total number of occurrences of the word “foo”:

```
let m1 = ...
let m2 = ...
let m3 = DelayMap.unionWith((x, y) -> x + y, m1, m2);
DelayMap.getWithDefault("foo", 0, m3) |> println
```

Here the `unionWith` function merges two maps using the supplied merge function to resolve conflicts when a key occurs in both maps. The merge operation is pure and hence `unionWith` is evaluated lazily. This means that we only have to merge and perform the addition for the key “foo” (and any other key we may look up).

@LazyWhenPure

```
def adjust(...)
def adjustWithKey(...)
def insertWith(...)
def insertWithKey(...)
def map(...)
def mapWithKey(...)
def unionWith(...)
def unionWithKey(...)
def update(...)
def updateWithKey(...)
```

(a) `DelayMap`: lazy when pure.

@ParallelWhenPure

```
def count(...)
def maximumKeyBy(...)
def maximumKeyBy(...)
def maximumValueBy(...)
def minimumKeyBy(...)
def minimumValueBy(...)
def sumWith(...)
def productWith(...)
```

(b) `DelayMap`: parallel when pure.

■ **Figure 5** Selective *Lazy* and *Parallel* datastructure, depending on purity of function arguments.

■ **Table 1** Overview of Data Structures. (*LWP* = Lazy When Pure, *PWP* = Parallel When Pure).

Data Structure	Lines	Tests	Functions	@LazyWhenPure	@ParallelWhenPure	LWP + PWP
Set	610	384	51	-	5	-
Map	924	591	80	-	9	-
DelayList	1,158	498	54	9	-	-
DelayMap	786	298	58	10	9	2

4.4.2 Example II

We can merge the two `DelayMaps`, while using a mutable list to compute all the words that occur in both maps:

```
let m1 = ...
let m2 = ...
let duplicates = MutList.empty();
let merge = (key, x, y) -> {
  MutList.add!(key, duplicates);
  x + y
};
let m3 = DelayMap.unionWithKey(merge, m1, m2)
```

The merge operation is impure and hence `unionWith` is evaluated eagerly. This ensures that the mutable list `duplicates` is updated correctly.

4.5 Summary

We have demonstrated the usefulness of `reifyEff` by using it in four data structures:

- We refactored the Flix Standard Library implementation of the `Set` and `Map` data structures to use selective parallel evaluation.
- We have introduced two new data structures: `DelayList` which uses selective lazy evaluation and `DelayMap` which uses selective lazy and parallel evaluation.

Table 1 shows an overview of the data structures that we have implemented. The `Data Structure` column gives the name of the data structure. The `Lines` column gives the number of Flix source code lines (excluding tests). The `Tests` column gives the number of manually written unit tests. The `Functions` column gives the number of functions implemented on the data structure. Most functions are first-order and “terminal”. For example, `Set.memberOf` is first-order and terminal, i.e., it does not transform the `Set` but rather returns a value. The `@LazyWhenPure` gives the number of functions that use purity reflection to enable lazy evaluation. The `@ParallelWhenPure` gives the number of functions that use purity reflection to enable parallel evaluation. The `LWP + PWP` gives the number of functions that use purity reflection to enable *both* lazy and parallel evaluation. For example, the line for `DelayMap` shows that the data structure is implemented in 786 lines of Flix code with 298 unit tests. The data structure offers 58 functions of which 10 use purity reflection to enable lazy evaluation, 7 use purity reflection to enable parallel evaluation, and 2 use purity reflection to enable both. Except for `DelayList`, these three data structures build on a Red-Black Tree, whose line counts are not included.

5 Compilation Strategies

We now discuss two ways to implement purity reflection: one based on runtime dispatch and the other on a novel form of effect-aware monomorphization. Both approaches rely on the type and effect system since we cannot readily determine if an expression will be pure at runtime. In other words, our extension of Flix only assigns meaning to well-typed programs.

5.1 Runtime Dispatch

Given a well-typed program, we can annotate each closure with a Boolean (or Boolean formula) to track if it is pure or impure. For a first-order function (i.e., a function without function arguments), its purity is readily determined by the typing judgment. For a higher-order function, its purity may depend on its function arguments. In this case, we label the closure with a Boolean formula that refers to the purity of the closure arguments. Given such an annotated closure, the `reifyEff` construct can be implemented by simply inspecting these annotations (and potentially evaluating a Boolean formula). For example, if we have the program fragment:

```
let f = x -> x + 1;
let g = x -> println(x);
let h = u -> v -> x -> u(v(x));
```

Then the closure of `f` stores a Boolean formula which is the constant true, the closure of `g` stores a Boolean formula which is the constant false, and the closure of `h` stores a Boolean formula which is the conjunction of the two bits of the higher-order arguments `u` and `v`. Thus, at runtime, the purity of `h` can be computed once `u` and `v` are known.

The cost of the runtime dispatch strategy is that we must store a Boolean formula with each closure. For first-order functions, this is just the constant true or false. For higher-order functions, it is a formula with several variables corresponding to its higher-order functions. Thus, in general, these formulas will be small since most functions are first-order and since higher-order functions tend to have only a few function arguments. Hence, the increase in code size should be modest. At runtime, the `reifyEff` construct has to evaluate small Boolean formulas which should be fast.

5.2 Effect-Aware Monomorphization

As an alternative to runtime dispatch, we propose an effect-aware form of monomorphization.

Monomorphization is a compile-time transformation that replaces polymorphic functions with copies that are specialized to the concrete types of their arguments. For example, if `List.map` is used with both integer and string lists, then monomorphization generates two copies of `List.map`: one specialized to integers and one specialized to strings. Monomorphization avoids boxing at the cost of larger executables. In practice, code size can be significantly reduced with the proper use of inlining and dead code elimination. A potential downside of monomorphization is that it may prevent separate compilation.

Before our work, the Flix compiler performed specialization of type variables and erased effect variables. In this work, we have extended the Flix compiler to specialize effect variables. In other words, the Flix compiler is now able to generate two versions of `List.map[Int]`: one specialized for pure functions and one specialized for impure functions. The upshot is that `reifyEff` can be eliminated – entirely at compile-time – because its argument is statically known to be pure or impure. Thus, the use of `reifyEff` incurs zero runtime overhead.

A technical detail is that during monomorphization a type or effect variable can potentially be left un-instantiated. For example, in the expression `Nil == Nil` each `Nil` can be given any type α . For effect variables, we can use two strategies to deal with such situations:

- **(Opportunistic)** We opportunistically treat all un-instantiated effect variables as pure. This is sound because the type system is closed under substitution (i.e., if a variable is free we may substitute it by \top).
- **(Conservative)** We reject programs that contain un-instantiated effect variables during monomorphization. The programmer can always resolve the situation with a type (or effect) annotation.

In our extension of Flix, we choose the conservative option because it is consistent with how ordinary un-instantiated type variables are treated. As an example, the following (contrived) program has an un-instantiated effect variable:

```
let f = g -> reifyEff(g) {
  case Pure(w) => g
  case -       => g
};
f(f)
```

Here the type of f is: $\forall \alpha_1, \alpha_2, \beta. (\alpha_1 \xrightarrow{\beta} \alpha_2) \xrightarrow{\top} (\alpha_1 \xrightarrow{\beta} \alpha_2)$. When f is applied to *itself* it returns a function of the same type which has an un-instantiated effect variable even after monomorphization. In Section 7, we investigate how common un-instantiated effect variables are in real programs. We have not observed un-instantiated effect variables in existing code.

5.3 Discussion

We believe that both the runtime dispatch and the effect-aware monomorphization approaches are viable. We decided to implement purity reflection via monomorphization since:

- Flix already uses monomorphization to eliminate parametric polymorphism.
- Monomorphization enables more aggressive optimizations performed by the Flix inliner.
- Monomorphization ensures that the technique imposes zero runtime overhead.

Finally, as our experiments in Section 7 demonstrate, the increase in compilation time and code size is small.

6 Implementation

We have implemented purity reflection as an extension of the Flix programming language.

6.1 The Flix Programming Language

Flix is a functional-first, imperative, and logic programming language that supports algebraic data types, pattern matching, higher-order functions, parametric polymorphism, type classes, higher-kinded types, polymorphic effects, extensible records, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination [24, 25, 26, 27, 28].

The Flix compiler project, including the standard library and tests, consists of 230,000 lines of Flix and Scala code. Adding `reifyEff` and effect-aware monomorphization required less than 2,000 lines of code. The extended Flix Standard Library required approximately 4,000 lines of code with unit tests (see Section 4).

Flix, with our extension, is open source, ready for use, and freely available at:

<https://flix.dev/> and <https://github.com/flix/flix/>

6.2 Integration with Type Classes

Flix supports type classes and higher-kinded types. The Flix compiler implements type classes using monomorphization, i.e., there is no dynamic dispatch or dictionary passing. This also has the advantage that purity reflection works with type classes without any modifications. For example, if a polymorphic function requires a `Foldable` instance because of a call to `Foldable.count` then during monomorphization the call to `Foldable.count` will be replaced by a call to the appropriate instance function. Thus, the benefits of selective lazy and/or parallel evaluation are available even for polymorphic functions that use type classes.

7 Evaluation

We now turn to the question of the viability of an implementation strategy based on effect-aware monomorphization. In particular, we consider the following research questions:

- **RQ1:** What is the impact of effect-aware monomorphization on compilation time?
- **RQ2:** What is the impact of effect-aware monomorphization on code size?
- **RQ3:** How common are un-instantiated effect variables in practice?

7.1 RQ1 and RQ2: Impact on compilation time and code size

Monomorphization specializes (i.e., copies) functions for each of their concrete type arguments. For example, the `List.map` function has the polymorphic type:

$$\forall a, \forall b, \forall e. (a \xrightarrow{e} b) \rightarrow \text{List}[a] \xrightarrow{e} \text{List}[b]$$

Under standard monomorphization, the `List.map` function is copied for every instantiation of the type variables a and b . If, for example, `List.map` is called with the two functions that have types: `Int32 → Bool` and `String → Int32`, we would get two copies of `List.map`.

With effect-aware monomorphization, a function is copied for every instantiation of its type variables *and* its effect variables. In other words, `List.map` will be copied for every instantiation of a , b , and e . Every effect variable is either pure or impure which means that we can get two copies of a function per effect variable (in addition to its other type variables). In the worst case, this can lead to an exponential blow-up in the number of copies.

We can construct a worst-case example with three effect variables:

```
def hof(f: a -> b & ef1, g: a -> b & ef2, h: a -> b & ef3): Unit = ...

def p(): Unit & Pure = ... // a pure function
def i(): Unit & Impure = ... // an impure function

def main(): Unit & Impure =
  hof(p, p, p);
  hof(i, p, p);
  // ... omitted for brevity ...
  hof(i, i, i)
```

The higher order function `hof` takes three function arguments f , g , and h . Each function argument has an effect variable, which can be instantiated to pure or impure. Inside `main`, we call `hof` with all possible combinations of pure and impure function arguments. There are $2^3 = 8$ combinations of these. This means that during effect-aware monomorphization we will construct 8 copies of `hof`, which duplicates its entire function body. If `hof` is large this can lead to a blow-up in compilation time and code size.

■ **Table 2** Impact of Effect-Aware Monomorphization on Compilation Time and Code Size.

†: The “DeliveryDate” and “Stratifier” programs depend on the Flix Datalog engine, which is a part of the Flix Standard Library and implemented in Flix itself. Hence, these programs are not actually 35-116 lines of code, but more accurately thought of as 35-116 lines of code *plus* the 3,055 lines of code used to implement the Datalog engine in Flix.

Program	Std. Monomorphization				Effect-Aware Monomorphization		
	Lines	Time	Bytes	Classes	Time	Bytes	Classes
Standard Library	33,689	4.6s	4,954	8	–	–	–
BoolTable	206	4.6s	766,638	759	4.6s (+0%)	766,546 (+0%)	759 (+0%)
DeliveryDate †	35	5.1s	2,913,210	2,762	5.1s (+0%)	3,019,959 (+4%)	2,888 (+5%)
fcwg	2,796	5.0s	1,700,982	1,911	5.0s (+0%)	1,699,107 (+0%)	1,911 (+0%)
flixball	1,767	4.8s	875,858	1,012	4.8s (+0%)	877,762 (+0%)	1,012 (+0%)
IfNoSub	1,870	5.2s	1,967,390	1,849	5.2s (+0%)	2,046,480 (+4%)	1,921 (+4%)
JSON	348	4.9s	486,521	573	4.9s (+0%)	487,025 (+0%)	573 (+0%)
Regex	1,891	4.5s	223,429	316	4.5s (+0%)	222,479 (+0%)	316 (+0%)
Stratifier †	116	4.9s	3,063,802	2,925	4.9s (+0%)	3,191,718 (+4%)	3,070 (+5%)
TestDelayList	3,060	5.1s	4,050,485	5,304	5.1s (+0%)	4,174,417 (+3%)	5,431 (+2%)
TestDelayMap	2,039	5.1s	4,111,046	5,090	5.1s (+0%)	4,750,187 (+16%)	5,631 (+11%)
TestMap	2,780	5.4s	5,230,078	5,782	5.4s (+0%)	5,572,129 (+7%)	6,107 (+6%)
TestSet	1,640	4.9s	2,598,250	2,759	4.9s (+0%)	2,757,388 (+6%)	2,930 (+6%)

Analysis

Given a polymorphic function f with n type parameters (quantified type variables) and m effect parameters (quantified Boolean variables), effect-aware monomorphization will create at most $t^n \times 2^m$ copies of f where t is the number of types that occur in the program after type checking but before monomorphization. In practice, type-based monomorphization does not lead to an exponential blow-up, but what about effect-aware monomorphization?

Table 2 shows the impact of effect-aware monomorphization on compilation time and code size for several Flix programs. We briefly discuss each program: The Flix “Standard Library” is included for completeness. When the library is compiled alone, without any entry point, a mere 8 Java classes are generated. These 8 classes are hard-coded and are always emitted. One class represents the Unit value. Other classes represent various exceptions. “BoolTable” is a Flix program to generate a table of smallest formulas for all Boolean functions of 4 arguments. “DeliveryDate” is a Flix program that uses first-class Datalog constraints with lattice semantics to compute the earliest delivery date for a “component” that consists of subcomponents, each with its delivery date and assembly time. “fcwg” is a Flix program generator that generates wrapper code for Java classes. “flixball” is a basic multi-player, 2-dimensional shooter game, run in the console. Bots compete in a last-player standing arena, taking simultaneous turns to rotate, move, or fire their weapon. “IfNoSub” is an implementation of Algorithm W for Flix written in Flix. It captures the relational nullable type system from [28]. “JSON” is, as the name implies, a JSON library for Flix. “Regex” is, as the name implies, a Regex library for Flix (based on Java’s regex package). “Stratifier” is a Flix program that uses first-class Datalog constraints with lattice semantics to implement a version of Ullman’s algorithm to compute the stratification of a Datalog program. “TestX” is the collection of unit tests for the data structure X .

We now explain each column of the table: The `Program` column gives the name of the Flix program. The `Lines` column shows the number of lines of code in the program (excluding the Flix Standard Library), The `Time` column shows the total compilation time in seconds (without effect-aware monomorphization). The `Bytes` column shows the number of bytes generated by the compiler (without effect-aware monomorphization). The `Classes` column shows the number of classes generated by the Flix compiler (without effect-aware monomorphization). The last three columns then repeat but now with effect-aware monomorphization. The numbers in parentheses show the percentage increase (resp. decrease) in the specific measurement.

For example, the “`IfNoSub`” program consists of 1,870 lines of code (in addition to the 30,000 lines of code from the Flix Standard Library). Under type-based monomorphization, the Flix compiler generates 1,849 Java classes, totaling 1,967,390 bytes in 5.2s. With effect-aware monomorphization, the compiler generates 1,921 Java classes totaling 2,046,480 bytes in the same amount of time. This represents a 4% increase in code and classes.

As a second example, the “`TestDelayMap`” program consists of 2,039 lines of code, which contain all the unit tests for the `DelayMap` data structure. Every purity reflective function is tested with both a pure and impure function argument. Under type-based monomorphization, the Flix compiler generates 5,090 Java classes totaling 4,111,046 bytes in 5.1s. With effect-aware monomorphization, the compiler generates 5,631 Java classes totaling 4,750,187 bytes within the same time. This represents an 16% increase in code and an 11% increase in classes.

As Table 2 shows, the cost of effect-aware monomorphization is low. For real programs, there is no increase in compile time and the increase in code size is between 0% to 5%.

We offer a few explanations why real programs show only a modest increase:

- Most functions are first-order (i.e., they do not take function arguments). A first-order function cannot be copied by effect-aware monomorphization, hence it cannot lead to increased compilation time or code size. To give two examples: In the `Set` module 17/45 functions are higher-order whereas in the `String` module 24/94 functions are higher-order.
- The majority of function calls are to first-order functions. For example, `List.sum` is probably more widely used than e.g., `List.zipWith3`. In other words, there are fewer higher-order functions than first-order functions, and they are on the balance also less likely to be used.
- When a higher-order function is used, it is not necessarily used with both pure and impure function arguments. For example, in many of the programs, higher-order functions are always used with a pure or an impure function argument, but more rarely with both.

In sum, we conclude that effect-aware monomorphization is a viable implementation strategy.

7.2 RQ3: How common are un-instantiated effect variables in practice?

As discussed in Section 5.2, an effect variable is potentially left un-instantiated during monomorphization. We showed a carefully crafted example – which relied on self-application – that would lead to an un-instantiated effect variable. We discussed two sound solutions: (i) an optimistic strategy that treats every un-instantiated variable as `true` (i.e., as pure), and (ii) a conservative strategy that rejects programs with un-instantiated (effect) variables. Flix uses the conservative strategy.

An important empirical question is then how common such situations are. In more than 100,000 lines of Flix code, we have never encountered a single un-instantiated effect. In fact, we have only been able to trigger the rejection with our carefully crafted example. We conclude that un-instantiated effect variables are not of practical concern. Intuitively, most expressions are either pure or impure. A few expressions are effect-polymorphic, but they are almost always called with pure or impure function arguments. Consequently, during monomorphization, we always end up with expressions that are either pure or impure.

8 Related Work

We consider related work along three axes: type and effect systems, reflection, and streams.

8.1 Type and Effect Systems

The Flix type and effect system is based on Hindley-Milner [12, 30, 7] extended with Boolean unification [29, 27]. The Flix system is effect-polymorphic, a notion that goes back to Lucassen et al. [23].

Algebraic effects is a hot research topic [33, 14, 19, 1, 21, 3, 4]. An algebraic effect system allows the programmer to define a collection of effects that can be invoked and interpreted by *effect handlers* installed on the stack (similar to exceptions). A type and effect system for an algebraic effect system ensures that all effects are ultimately handled. Most prototype programming languages that support algebraic effects *and* complete type inference are based on row polymorphism. Purity reflection seems orthogonal to algebraic effects; we are not interested in a collection of effects nor in how to interpret them. We are interested in enabling higher-order functions to selectively use lazy or parallel evaluation when passed pure function arguments. As interesting future work, we can imagine a type and effect system that tries to combine algebraic effects with purity reflection while retaining complete type inference.

A line of research has used uniqueness and ownership type systems to prevent data races and deadlocks and to enable parallelism [2, 6, 9]. Boyapati et al. present a type system that prevents data races and deadlocks. In the system, programmers partition locks into equivalence classes and define a partial order on them. The type checker then verifies that the locks are acquired in descending order [2]. Craik and Kelly present a type, effect, and ownership system that uses read-and-write effect sets to reason about data dependence. This information is then exploited to enable data or task parallelism [6]. Gordon et al. present a type and effect system that restricts updates to mutable memory shared by multiple threads. The system relies on a combination of immutable and uniqueness types, which ensure the absence of data races [9].

8.2 Type Case and Effect Reflection

Tarditi et al. propose *type case*, a meta-programming construct that enables polymorphic functions to reflect on their concrete type arguments [11, 36]. In the TIL Standard ML compiler, type casing is used to implement several polymorphic functions more efficiently. For example, an array indexing (“subscript”) operation can be implemented more efficiently if the compiler knows the concrete type of the underlying array. One might think of our work as an *effect case* which is used to enable selective lazy or parallel evaluation.

Long et al. propose a calculus and type system with reflection for effects representing region accesses [22]. Their effects are first-class expressions that can be inspected by pattern matching. The features of their system are orthogonal to our system: they have a hybrid approach, based on static and dynamic types; their calculus provides over-approximating (may) and under-approximating (must) types, and is based on sub-typing/effecting and refinement types. Instead, we support type and effect polymorphism with inference based on Boolean unification. We also provide an implementation in a programming language.

8.3 Streams

Broadly speaking, the relation between our work and work on streams is that most stream implementations aim to provide lazy and/or parallel evaluation capabilities in programming languages that are eager and impure, at the risk of unsoundness. Purity reflection allows library authors to soundly determine *when* it is safe to use lazy and/or parallel evaluation.

Wadler et al. introduce the notion of “odd” and “even” lazy lists [38]. Wadler et al.’s observation is that in the standard definition of a lazy list, the first element of the list is eager whereas the rest of the list is lazy. In other words, the lazy list is not entirely lazy which can lead to unexpected behavior. We agree with this observation and in our real implementation of `DelayList` every element is lazy.

Jones et al. present an extension of Haskell, where the programmer can define a collection of rewrite rules that are applied to the program by the Haskell compiler [13]. Using rewrite rules, a programmer (typically a library author) can express program optimizations as a collection of transformations. The mechanism is extended and applied to stream fusion in [5]. The rewrite mechanism could provide an alternative implementation of purity reflection. Similar to our solution using monomorphization, rewrite rules are applied at compile time and induce no run-time overhead. However, we see two advantages in our solution based on purity reflection: First, rewrite rules only apply if there is a syntactic pattern match, while our technique also applies across various let-bindings or even function calls, since it is based on a type and effect system that propagates information. Second, a programmer could easily add unsound rewrite rules, while our type system provides some guarantees; in particular, impure functions cannot be postponed.

Prokopec et al. show that the overhead of functional combinators (e.g., `filter`, `map`) on the JVM can be overcome with a sufficiently aggressive JIT compiler [34]. Kiselyov et al. present a technique to overcome the overhead of stream operations through staging [18]. Møller and Veileborg propose to use static analysis to eliminate the overhead of stream pipelines in Java [32]. In summary, the bulk of this work has focused on *how* to make streams execute faster. In contrast, our work concerns *when* it is safe to do so. We use monomorphization to implement purity reflection. After monomorphization, we have a mono-typed program where each use of a non-terminal operation (e.g., `filter` and `map`) has been replaced by its eager or lazy version. We can pass the monomorphed AST to any technique that performs stream fusion without the risk of unsoundness.

Khatchadourian et al. present a study on the use and misuse of streams in Java [17]. Interestingly two of their findings are: “Finding 1: Stream parallelization is not widely used” and “Finding 3: Streams tend not to have side effects.” The former finding could suggest that even though parallel streams are readily available, developers are either unaware or reluctant to use them. With purity reflection, the choice of whether to use parallelism rests not just with the programmer but also with the library author. This suggests that purity reflection can help programmers by exploiting parallelism when they did not consider it themselves.

Khatchadourian et al. also present an automatic technique to refactor code to use streams more efficiently [16]. The technique is based on heavy-weight program analysis. Flix, with purity reflection, offers an alternative approach where reasoning about and reflecting on purity is built directly into the language.

9 Conclusion

We have proposed *purity reflection*, a new programming language feature that enables higher-order functions to vary their behavior depending on the purity of their function arguments. Purity reflection enables selective use of lazy and/or parallel evaluation, while ensuring that side effects are never lost or re-ordered. We have implemented purity reflection in the Flix programming language. We have retrofitted and extended the Flix Standard Library with new data structures that automatically use lazy or parallel evaluation when it is safe to do so. Effect-aware monomorphization provides a mechanism to implement purity reflection as

a construct that is entirely eliminated at compile-time. Therefore, the technique imposes no run-time overhead. Experimental results show that the cost of effect-aware monomorphization in compilation time and code size is minimal.

References

- 1 Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1), 2015. doi:10.1016/j.jlamp.2014.02.001.
- 2 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002. doi:10.1145/582419.582440.
- 3 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effect Handlers for the Masses. *Proc. of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276481.
- 4 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428194.
- 5 Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. *International Conference of Functional Programming (ICFP)*, 42(9), 2007. doi:10.1145/1291220.1291199.
- 6 Andrew Craik and Wayne Kelly. Using Ownership to Reason about Inherent Parallelism in Object-Oriented Programs. In *International Conference on Compiler Construction (CC)*, 2010. doi:10.1007/978-3-642-11970-5_9.
- 7 Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, The University of Edinburgh, 1984.
- 8 Jacques Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*, 2004. doi:10.1007/978-3-540-24754-8_15.
- 9 Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012. doi:10.1145/2398857.2384619.
- 10 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, 2002. doi:10.1145/512529.512563.
- 11 Robert Harper and Greg Morrisett. Compiling Polymorphism using Intensional Type Analysis. In *Principles of Programming Languages (POPL)*, 1995. doi:10.1145/199448.199475.
- 12 Roger Hindley. The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society (AMS)*, 1969.
- 13 Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. In *Haskell Workshop*, 2001.
- 14 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in Action. *International Conference on Functional Programming (ICFP)*, 48(9), 2013. doi:10.1145/2544174.2500590.
- 15 Ohad Kammar and Gordon D. Plotkin. Algebraic Foundations for Effect-dependent Optimisations. In *Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103656.2103698.
- 16 Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. *Science of Computer Programming*, 2020. doi:10.1016/j.scico.2020.102476.

- 17 Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. An Empirical Study on the Use and Misuse of Java 8 Streams. In *Fundamental Approaches to Software Engineering (FASE)*, 2020. doi:10.1007/978-3-030-45234-6_5.
- 18 Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream Fusion, to Completeness. In *Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009880.
- 19 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible Effects: An Alternative to Monad Transformers. *Haskell Workshop*, 2013. doi:10.1145/2578854.2503791.
- 20 Daan Leijen. Extensible Records with Scoped Labels. *Trends in Functional Programming (TFP)*, 2005.
- 21 Daan Leijen. Type Directed Compilation of Row-typed Algebraic Effects. In *Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009872.
- 22 Yuheng Long, Yu David Liu, and Hridesh Rajan. First-class Effect Reflection for Effect-Guided Programming. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016. doi:10.1145/3022671.2984037.
- 23 John M Lucassen and David K Gifford. Polymorphic Effect Systems. In *Principles of Programming Languages (POPL)*, 1988.
- 24 Magnus Madsen. The Principles of the Flix Programming Language. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2022. doi:10.1145/3563835.3567661.
- 25 Magnus Madsen and Ondřej Lhoták. Fixpoints for the Masses: Programming with First-class Datalog Constraints. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428193.
- 26 Magnus Madsen, Jonathan Lindegaard Starup, and Ondřej Lhoták. Flix: A Meta Programming Language for Datalog. In *Proc. International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0 2022)*, 2022.
- 27 Magnus Madsen and Jaco van de Pol. Polymorphic Types and Effects with Boolean Unification. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428222.
- 28 Magnus Madsen and Jaco van de Pol. Relational Nullable Types with Boolean Unification. *Proc. of the ACM on Programming Languages*, 5(OOPSLA), 2021. doi:10.1145/3485487.
- 29 Urusula Martin and Tobias Nipkow. Boolean Unification - The Story So Far. *Journal of Symbolic Computation*, 1989.
- 30 Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 1978.
- 31 Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media, 2013.
- 32 Anders Møller and Oskar Haarklou Veileborg. Eliminating Abstraction Overhead of Java Stream Pipelines using Ahead-of-Time Program Optimization. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428236.
- 33 Matija Pretnar and Gordon D Plotkin. Handling Algebraic Effects. *Logical Methods in Computer Science*, 2013. doi:10.2168/LMCS-9(4:23)2013.
- 34 Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making Collection Operations Optimal with Aggressive JIT Compilation. In *Proc. International Symposium on Scala*, 2017. doi:10.1145/3136000.3136002.
- 35 John C Reynolds. The meaning of types from intrinsic to extrinsic semantics. *BRICS Report Series*, 7(32), 2000.
- 36 David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A Type-directed Optimizing Compiler for ML. In *Programming Language Design and Implementation (PLDI)*, 1996. doi:10.1145/249069.231414.
- 37 Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Information and Computation*, 1997. doi:10.1006/inco.1996.2613.

- 38 Philip Wadler, Walid Taha, and David MacQueen. How to Add Laziness to a Strict Language Without Even Being Odd. In *SML'98, The SML workshop*, 1998.
- 39 Mitchell Wand. Complete Type Inference for Simple Objects. In *Logic in Computer Science*, 1987.
- 40 Keith Wansbrough and Simon L. Peyton Jones. Once Upon a Polymorphic Type. In *Principles of Programming Languages (POPL)*, 1999. doi:10.1145/292540.292545.

Exact Separation Logic

Towards Bridging the Gap Between Verification and Bug-Finding

Petar Maksimović

Imperial College London, UK

Runtime Verification Inc., Urbana, IL, USA

Caroline Cronjäger

Ruhr-Universität Bochum, Germany

Andreas Löw

Imperial College London, UK

Julian Sutherland

Nethermind, London, UK

Philippa Gardner

Imperial College London, UK

Abstract

Over-approximating (OX) program logics, such as separation logic (SL), are used for *verifying* properties of heap-manipulating programs: all terminating behaviour is characterised, but established results and errors need not be reachable. OX function specifications are thus incompatible with true bug-finding supported by symbolic execution tools such as Pulse and Pulse-X. In contrast, under-approximating (UX) program logics, such as incorrectness separation logic, are used to *find* true results and bugs: established results and errors are reachable, but there is no mechanism for understanding if all terminating behaviour has been characterised.

We introduce exact separation logic (ESL), which provides fully-verified function specifications compatible with both OX verification and UX true bug-finding: all terminating behaviour is characterised and all established results and errors are reachable. We prove soundness for ESL with mutually recursive functions, demonstrating, for the first time, function compositionality for a UX logic. We show that UX program logics require subtle definitions of internal and external function specifications compared with the familiar definitions of OX logics. We investigate the expressivity of ESL and, for the first time, explore the role of abstraction in UX reasoning by verifying abstract ESL specifications of various data-structure algorithms. In doing so, we highlight the difference between *abstraction* (hiding information) and *over-approximation* (losing information). Our findings demonstrate that abstraction cannot be used as freely in UX logics as in OX logics, but also that it should be feasible to use ESL to provide tractable function specifications for self-contained, critical code, which would then be used for both verification and true bug-finding.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Program reasoning; Theory of computation → Separation logic; Theory of computation → Hoare logic; Theory of computation → Abstraction

Keywords and phrases Separation logic, program correctness, program incorrectness, abstraction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.19

Related Version *Extended Version*: <https://arxiv.org/abs/2208.07200>

Funding Maksimović, Löw and Gardner were partially supported by the EPSRC Fellowship “VetSpec: Verified Trustworthy Software Specification” (EP/R034567/1). Cronjäger was partially supported by the Erasmus Plus Student Mobility for Traineeships scheme.

Acknowledgements We would like to thank Sacha-Élie Ayoun and Daniele Nantes Sobrinho for the many discussions that have improved the quality of the paper. We would also like to thank the anonymous reviewers for their comments.



© Petar Maksimović, Caroline Cronjäger, Andreas Löw, Julian Sutherland, and Philippa Gardner; licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 19; pp. 19:1–19:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Over-approximating (OX) program logics were introduced to reason about program correctness, starting with Hoare logic [18] and evolving to separation logic (SL) [27, 30]. SL is used for *verification* and features function specifications of the form $\{P\} f(\vec{x}) \{Q\}$, the meaning of which is that all terminating executions of the function f that start from a state in the pre-condition P end in a state covered by the post-condition Q . SL has the standard rule of *forward consequence*, which allows one to *lose information* (for example, if we had a post-condition with $x = 42$, we could soundly weaken this precise information to the less precise $x > 0$ or even to the non-informative `true`). In essence, the philosophy underlying the OX approach in general can be stated as:

no paths can be cut, but information can be lost.

A key property of SL is that function specifications are *compositional*, enabling *scalable* reasoning about the heap. This is due to their *locality*, which allows the pre-condition to describe only the partial state sufficient for the function to execute, and the *frame* property, which allows the function to be called in any larger state. SL function specifications have been used for verification of complex, real-world code in tools such as VeriFast [19], Iris [20], and Gillian [10, 23]. However, given that their post-conditions may describe states that are not reachable from their pre-conditions, such OX specifications are not compatible with true bug-finding, as found, for example, in Meta’s Pulse [28] and Pulse-X [21] tools.

Under-approximating (UX) program logics were recently introduced, originating from reverse Hoare logic (RHL) [8] for reasoning about correctness of probabilistic programs, and coming to prominence with incorrectness logic [26] and incorrectness separation logic (ISL) [28], which identified their bug-finding potential. ISL function specifications are of the form $[P] f(\vec{x}) [ok : Q_{ok}]$ and $[P] f(\vec{x}) [err : Q_{err}]$, the meaning of which is that any state in the success post-condition Q_{ok} or the error post-condition Q_{err} is reachable from some state in the pre-condition P by executing the function f ; this guarantees that all results and bugs reported in the post-conditions will be true. In contrast to SL, ISL uses the rule of *backward consequence*, which allows one to *cut paths* (for example, if we had a post-condition with $x > 0$, we could soundly strengthen this information to consider only the path in which $x = 42$). Therefore, the philosophy underlying UX logics in general can be summarised as:

paths can be cut, but no information can be lost.

When it comes to the use of ISL function specifications, whilst this has been implemented in Pulse-X, as far as we are aware, ISL does not feature function-call rules, and function compositionality for ISL and UX logics has not been proven. Moreover, as it is not possible to determine if UX specifications cover all terminating behaviour, they remain incompatible with verification and cannot therefore be used in tools such as VeriFast, Iris, and Gillian.

Our challenge is to develop a program logic in which we can state and prove function specifications that are compatible with *both* verification and true bug-finding. Our motivation comes from the unique flexibility and expressivity that such specifications would provide, as they could be used by verification and bug-finding tools alike, closing the gap between these two contrasting paradigms. From our experience in program logics and associated tool-building, we believe that the main use case for exact specification should be self-contained, critical code, such as widely-used data-structure libraries.

We introduce exact separation logic (ESL), with *exact* (EX) function specifications of the form $(P) f(\vec{x}) (ok : Q_{ok}) (err : Q_{err})$, whose meaning combines that of SL and ISL specifications: all terminating executions of the function that start from a state in the

pre-condition P end in a state covered by the post-conditions; *and* all states in two post-conditions are reachable from a state in the pre-condition by executing the function. The exactness of ESL can be captured by the slogan:

no paths can be cut and no information can be lost.

The slogan is supported by the rule of *equivalence*, which combines the forward consequence of SL and the backward consequence of ISL. In fact, ESL proof rules form a common core of SL and ISL, and ESL should therefore be a familiar setting to those acquainted with either.

We prove soundness for ESL with mutually recursive functions, which we believe is the first proof of function compositionality for a UX logic, and which transfers immediately to ISL. In doing so, drawing inspiration from InsecSL [25], we provide formal definitions of *external* and *internal* function specifications, which describe, respectively, the interface a function exposes towards its clients and towards its implementation, and highlight the difference in complexity between these two types of specifications in OX and UX reasoning.

Using numerous examples, we demonstrate here and in the extended version [24] how ESL can be used to reason about data-structure libraries, language errors, mutual recursion, and non-termination. In doing so, we introduce, for the first time, abstract predicates to UX reasoning and provide abstract function specifications for a number of data-structure algorithms, focussing on singly-linked lists and binary trees. In doing so, we highlight an important difference between the concepts of abstraction and over-approximation: in particular, abstraction corresponds to *hiding* information whereas over-approximation corresponds to *losing* it. Our findings demonstrate that, while abstraction cannot be used as freely in UX logics as in OX logics, sometimes resulting in less abstract specifications and more complex proofs, it should be feasible to use ESL to provide tractable function specifications for self-contained, critical code that can then be used for both verification and true bug-finding.

2 Exact Separation Logic by Example

We guide the reader through what it means to write ESL specifications and proofs by intuition and example, contrasting our findings with those known from SL and ISL.

Illustrative Example. Consider the command $C \triangleq \text{if } (x > 0) \{y := 42\} \text{ else } \{y := 21\}$, which can be specified, starting from the pre-condition $x \in \mathbb{Z}$, in ESL, SL, and ISL as follows:

<pre>(x ∈ ℤ) if (x > 0) { (x > 0) y := 42 (Q₁ : x > 0 ∧ y = 42) } else { (x ≤ 0) y := 21 (Q₂ : x ≤ 0 ∧ y = 21) } (Q₁ ∨ Q₂)</pre>	<pre>{ x ∈ ℤ } if (x > 0) { ... // Same as ESL ... } { Q₁ ∨ Q₂ } // Losing information { x ∈ ℤ ∧ y > 0 }</pre>	<pre>[x ∈ ℤ] if (x > 0) { [x > 0] y := 42 [Q₁ : x > 0 ∧ y = 42] } else { y := 21 } // Path cutting [x > 0 ∧ y = 42]</pre>
---	--	--

As ESL specifications must neither cut paths nor lose information (in this example, about the values of x and y), the ESL post-condition of C must be equivalent to $(x > 0 \wedge y = 42) \vee (x \leq 0 \wedge y = 21)$. In SL, it is possible to use forward consequence to weaken this

information and obtain, for example, $x \in \mathbb{Z} \wedge y > 0$, or just $x \in \mathbb{Z}$, or even just true . In ISL, it is possible to cut, for example, the **else** branch of the **if** statement, but the values of x and y must be maintained in the post-condition of the **then** branch, $x > 0 \wedge y = 42$.

One question that we have been often asked is whether it is simpler to prove an exact specification $(P) \text{ C } (ok : Q_{ok}) (err : Q_{err})$ in ESL, or to prove it separately in SL and ISL. The answer is that it is simpler to prove the specification in ESL. If a specification is exact, then it does not cut paths and it does not lose information. Therefore, the tools that make SL and ISL proofs simpler than ESL proofs, namely forward consequence and backward consequence, can only be used in very limited ways, if at all. From our experience, the ISL proof of an exact specification will turn out to be almost identical to the ESL one, and an SL proof on top of that would duplicate a large part of the work. In fact, if one were to try to prove the exact specification $(x \in \mathbb{Z}) \text{ C } ((x > 0 \wedge y = 42) \vee (x \leq 0 \wedge y = 21))$ from the above example in either SL or ISL, they would obtain exactly the same proof as in ESL.

We also emphasise that ESL is not meant to replace either SL or ISL. If one is interested in only verification or only bug-finding, then one should use a formalism tailored to that type of analysis to exploit the available shortcuts. However, if one wanted to use the same codebase for both verification and bug-finding, then ESL offers a way of providing specifications useful for both. One example of such a codebase would be a widely-used data-structure library, where some of the users use it for verification and others for bug-finding.

List-length in ESL: Intuition. We consider a list-length function, $\text{LLen}(x)$, which takes a list at x , does not modify it, and returns its length, and the following ESL specification:

$$(x = x \star \text{list}(x, n)) \text{ LLen}(x) (ok : \text{list}(x, n) \star \text{ret} = n)$$

This specification uses a standard list-length predicate, $\text{list}(x, n)$, which states that the length of the list at x equals n and is defined as follows:

$$\text{list}(x, n) \triangleq (x = \text{null} \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star \text{list}(x', n - 1)),$$

hiding the information about the values and internal node addresses of the list. Before proving this specification, we establish some intuition about why it holds. Let us assume that it does not hold and try to find a counter-example: by the meaning of ESL specifications, it is either not OX-valid or it is not UX-valid. The former, however, is not possible, as the analogous SL specification holds. The latter means that it is possible to find a state in the post-condition not reachable by the execution of f from any state in the pre-condition, and may be unfamiliar to the reader as UX program logics have been introduced only recently.

We start looking for such a state in the post-condition (post-model) by choosing some values for x and n : say, $x = 0$ and $n = 2$. This also fixes ret to 2. Then, we fully unfold $\text{list}(0, 2)$ to obtain $\exists v_1, x_1, v_2. 0 \mapsto v_1, x_1 \star x_1 \mapsto v_2, \text{null}$, and instantiate the existentials v_1 , x_1 , and v_2 : say, with 1, 4, and 9, respectively. In this way, we obtain the state described by the assertion $0 \mapsto 1, 4 \star 4 \mapsto 9, \text{null}$. When it comes to the pre-condition, x and n (and also x) are fixed by the post-model choices, and when we unfold the list, the pre-condition becomes $x = 0 \star \exists v_1, x_1, v_2. 0 \mapsto v_1, x_1 \star x_1 \mapsto v_2, \text{null}$. As the algorithm does not modify the list, it becomes clear that if we choose v_1 , x_1 , and v_2 as for the post-model (that is, 1, 4, and 9, respectively), the algorithm will reach our post-model. Given that the same reasoning would apply for any choice of x and n , we realise that the given specification is, in fact, also UX-valid and hence exact. This reveals an important observation, which is that

*abstraction does not always equate to over-approximation, that is,
hiding information does not always mean losing information.*

For those used to OX reasoning, it might appear that the post-condition $\text{list}(x, n) \star \text{ret} = n$ loses information about the structure of the list, but the insight here is that this information was never known in the pre-condition in the first place, as we also only had $\text{list}(x, n)$ there.

List-length in ESL: Proof Sketch. Reasoning about function specifications in the UX/EX setting has not been studied previously and requires subtle definitions of *external function specifications*, which provide the interface that the function exposes to the client, and *internal function specifications*, which provide the interface to the function's implementation. With OX logics, these are well-understood and the gap between them is small. For UX/EX logics, this gap is larger. We illustrate these concepts informally using the list-length example, and give the corresponding formal definitions in §4.

The proof sketch of the ESL external specification of the list-length algorithm is given in Figure 1. It is more complex than its SL counterpart (cf. [24]), but is manageable and comes with the benefit that this ESL specification can be used for both verification and bug-finding.

First, as the function is recursive, we have to provide a measure and prove the specification extended with this measure: in this case, the measure is $\alpha = n$, given by the length of the list. This measure is necessary to ensure the finite reachability property for mutually recursive functions in UX logics, and is a known technique from the work on total correctness specifications for OX logics [7, 9]. Recursive function calls are then allowed only if they use specifications of a strictly smaller measure, represented in the proof sketch by the function specification context $\Gamma(\alpha)$, which contains the specification of $\text{LLen}(x)$ for all $\beta < \alpha$.

The move from the external to the internal pre-condition initialises the local function variables to `null`. The ESL rule for the `if` statement, just like in SL, adds the condition to the then-branch, its negation to the else-branch, and collects the branch post-conditions using disjunction. The rules for the basic commands (here, the assignments $r := 0$ and $r := r + 1$ and the lookup $x := [x + 1]$) are also the same as in SL, as these are already exact. The unfolding of the list is also done in the same way, as unfolding always preserves equivalence; note how the condition of the `if` statement determines the appropriate disjunct for the list predicate. The recursive function call is allowed to go through as it is used with measure $n - 1$ (with the parts of the assertion representing the pre- and the post-condition highlighted).

The major difference between ISL/ESL and SL proofs is that we cannot lose information about the function parameters and local variables in the middle of the former. Therefore, we cannot simplify the assertions Q'_1 and Q'_2 further and cannot fold back the list predicate within the internal specification, as we would do in SL (cf. corresponding proof in [24]).

The most complex part of the proof sketch is the transition from the internal to the external post-condition, in which we have to somehow forget the local variables of the function, given that they must not spill out into the calling context. This is done by replacing them with fresh, existentially quantified logical variables, which in this case also allows us to use equivalence to fold back the list predicate and reach the target post-condition. The details of this transition, in which we denote $\text{ret} = n \star \alpha = n$ by R , are as follows:

$$\begin{aligned}
& \exists x_q, r_q. Q'[x_q/x][r_q/r] \star \text{ret} = r[x_q/x][r_q/r] \\
& \Leftrightarrow ((x = \text{null} \star n = 0) \vee (\exists x_q, r_q, v, x'. x_q = x' \star x \mapsto v, x' \star \text{list}(x', n - 1) \star r_q = n)) \star R \\
& \Leftrightarrow ((x = \text{null} \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star \text{list}(x', n - 1))) \star R \text{ // can fold now} \\
& \Leftrightarrow \text{list}(x, n) \star (n = 0 \vee n > 0) \star R \\
& \Leftrightarrow \text{list}(x, n) \star \text{ret} = n \star \alpha = n
\end{aligned}$$

Observe that, since we are proving an EX specification, we are not allowed to cut paths. This means that the ISL proof of the analogous ISL specification of $\text{LLen}(x)$ would be identical, noting that the use of equivalence would technically be replaced by backward consequence.

```

// Function is recursive and requires a measure:  $\alpha = n$ 
 $\Gamma(\alpha) \vdash (x = x \star \text{list}(x, n) \star \alpha = n)$ 
  LLen( $x$ ) {
    // Transition from external to internal pre-condition: initialise locals to null
    ( $x = x \star \text{list}(x, n) \star \alpha = n \star r = \text{null}$ )
    if ( $x = \text{null}$ ) {
      ( $x = x \star \text{list}(x, n) \star \alpha = n \star r = \text{null} \star x = \text{null}$ )
       $r := 0$ 
      ( $Q'_1 : x = x \star \text{list}(x, n) \star \alpha = n \star r = 0 \star x = \text{null}$ )
    } else {
      ( $x = x \star \text{list}(x, n) \star \alpha = n \star r = \text{null} \star x \neq \text{null}$ )
      // Unfold  $\text{list}(x, n)$  using the equivalence
      //  $\models \text{list}(x, n) \star x \neq \text{null} \Leftrightarrow \exists v, x'. x \mapsto v, x' \star \text{list}(x', n - 1)$ 
      ( $\exists v, x'. x = x \star x \mapsto v, x' \star \text{list}(x', n - 1) \star \alpha = n \star r = \text{null}$ )
       $x := [x + 1]$ ;
      ( $\exists v, x'. x = x' \star x \mapsto v, x' \star \text{list}(x', n - 1) \star \alpha = n \star r = \text{null}$ )
      // As  $\alpha - 1 < \alpha$ , we can use the specification of LLen( $x$ ) with measure  $\alpha - 1$ 
      ( $\exists v, x'. x = x' \star x \mapsto v, x' \star \text{list}(x', n - 1) \star \alpha - 1 = n - 1 \star r = \text{null}$ )
       $r := \text{LLen}(x)$ ;
      ( $\exists v, x'. x = x' \star x \mapsto v, x' \star \text{list}(x', n - 1) \star \alpha - 1 = n - 1 \star r = n - 1$ )
       $r := r + 1$ 
      ( $Q'_2 : \exists v, x'. x = x' \star x \mapsto v, x' \star \text{list}(x', n - 1) \star \alpha - 1 = n - 1 \star r = n$ )
    };
    ( $Q' : Q'_1 \vee Q'_2$ )
    return  $r$ 
    ( $Q' \star \text{ret} = r$ )
    // Transition from internal to external post-condition given in text
  }
  ( $\text{list}(x, n) \star \text{ret} = n \star \alpha = n$ )

```

■ **Figure 1** ESL proof sketch: $(x = x \star \text{list}(x, n)) \text{ LLen}(x) \text{ (ok : list}(x, n) \star \text{ret} = n)$.

List-insert in ESL: Intuition. The list-length function, $\text{LLen}(x)$, is an example of an algorithm where the EX specification is analogous to the traditional OX specification. At times, however, ESL specifications have to be more complex. Consider, for example, the list-insert algorithm $\text{LInsertFirst}(x, v)$, which inserts the element v at the beginning of the list x . Its traditional OX specification is:

$$\{x = x \star v = v \star \text{list}(x, vs)\} \text{ LInsertFirst}(x, v) \{ \text{list}(\text{ret}, v : vs) \}$$

where $\text{list}(x, vs)$ is the standard list predicate that exposes the values of the list:

$$\text{list}(x, vs) \triangleq (x = \text{null} \star vs = []) \vee (\exists v, x', vs'. x \mapsto v, x' \star \text{list}(x', vs') \star vs = v : vs')$$

Using the counter-example approach to check if this specification is EX-valid, we easily see that it loses information: in particular, no end-state where x is not the *second* pointer in the returned list ret is reachable from the given pre-condition. Consequently, for EX validity, we are required to use the following, less abstract, ESL specification for LInsertFirst :

$$(x = x \star v = v \star \text{list}(x, xs, vs)) \text{ LInsertFirst}(x, v) (\text{list}(\text{ret}, \text{ret} : xs, v : vs) \star \text{listHead}(x, xs))$$

where $\text{list}(x, xs, vs)$ is a predicate that exposes the internal pointers of a given list in addition to the values, and $\text{listHead}(x, xs)$ states that the list xs starts with x .

Further Examples. In §5 and [24], we give many additional examples of ESL specifications and proofs to illustrate reasoning about list algorithms and binary trees, as well as language errors, mutual recursion, non-termination, and client programs.

3 The Programming Language

We introduce ESL using a simple programming language, the syntax of which is given below.

Language Syntax

$$\begin{array}{l}
 v \in \text{Val} ::= n \in \text{Nat} \mid b \in \text{Bool} \mid s \in \text{Str} \mid \text{null} \mid \bar{v} \quad x \in \text{PVar} \\
 E \in \text{PExp} ::= v \mid x \mid E + E \mid E - E \mid \dots \mid E = E \mid E < E \mid \neg E \mid E \wedge E \mid \dots \mid E : E \mid E @ E \mid \dots \\
 C \in \text{Cmd} ::= \text{skip} \mid x := E \mid x := \text{nondet} \mid \text{error}(E) \mid \text{if } (E) C \text{ else } C \mid \text{while } (E) C \mid C ; C \mid \\
 \quad y := f(\bar{E}) \mid x := [E] \mid [E] := E \mid x := \text{new}(E) \mid \text{free}(E)
 \end{array}$$

Values, $v \in \text{Val}$, include: natural numbers, $n \in \text{Nat}$; Booleans, $b \in \text{Bool} \triangleq \{\text{true}, \text{false}\}$; strings, $s \in \text{Str}$; a dedicated value `null`; and lists of values, $\bar{v} \in \text{List}$. Expressions, $E \in \text{PExp}$, comprise values, program variables, $x \in \text{PVar}$, and various unary and binary operators (e.g., addition, equality, negation, conjunction, list prepending, and list concatenation). Commands comprise: the variable assignment; non-deterministic number generation; error raising; the `if` statement; the `while` loop; command sequencing; function call; and memory management commands, that is, lookup, mutation, allocation, and deallocation. The sets of program variables for expressions and commands, denoted by $\text{pv}(E)$ and $\text{pv}(C)$ respectively, and the sets of modified variables for commands, denoted by $\text{mod}(C)$, are defined in the standard way.

► **Definition 1 (Functions).** A function, denoted by $f(\bar{x}) \{ C; \text{return } E \}$, comprises: a function identifier, $f \in \text{Str}$; the function parameters, \bar{x} , given by a list of distinct program variables; a function body, $C \in \text{Cmd}$; and a return expression, $E \in \text{PExp}$, with $\text{pv}(E) \subseteq \{\bar{x}\} \cup \text{pv}(C)$.

Program variables in function bodies that are not the function parameters are treated as local variables initialised to `null`, with their scope not extending beyond the function.

► **Definition 2 (Function Implementation Contexts).** A function implementation context, $\gamma : \text{Str} \rightarrow_{\text{fin}} \text{PVar List} \times \text{Cmd} \times \text{PExp}$, is a finite partial function from function identifiers to their implementations. For $\gamma(f) = (\bar{x}, C, E)$, we also write $f(\bar{x}) \{ C; \text{return } E \} \in \gamma$.

We next define an operational semantics that gives a complete account of the behaviour of commands and does not get stuck on any input, as we explicitly account for language errors and missing resource errors.

► **Definition 3 (Stores, Heaps, States).** Variable stores, $s : \text{PVar} \rightarrow_{\text{fin}} \text{Val}$, are partial finite functions from program variables to values. Heaps, $h : \text{Nat} \rightarrow_{\text{fin}} (\text{Val} \uplus \emptyset)$, are partial finite functions from natural numbers to values extended with a dedicated symbol $\emptyset \notin \text{Val}$. Program states, $\sigma = (s, h)$, consist of a store and a heap.

Heaps are used to model the memory, and the dedicated symbol $\emptyset \notin \text{Val}$ is required for UX frame preservation¹ to hold (cf. Definition 10). In particular, $h(n) = v$ means that an allocated heap cell with address n contains the value v ; and $h(n) = \emptyset$ means that a heap

¹ UX frame preservation means that if a program runs with a non-missing outcome to a given final state, then it also runs with the same outcome to an extended final state, with the extension (the *frame*) unaffected by the execution. From ISL [28], it is known that losing deallocation information breaks UX frame preservation; the solution is to keep track of deallocated cells, which we achieve by using \emptyset .

cell with address n has been deallocated [11–14, 28]. This linear memory model is used in much of the SL literature, including ISL [28]. Onward, \emptyset denotes the empty heap, $h_1 \uplus h_2$ denotes heap disjoint union, and $h_1 \# h_2$ denotes that h_1 and h_2 are disjoint.

► **Definition 4** (Expression Evaluation). *The evaluation of an expression E with respect to a store s , denoted $\llbracket E \rrbracket_s$, results in either a value or a dedicated symbol denoting an evaluation error, $\dagger \notin \text{Val}$. Some illustrative cases are:*

$$\llbracket v \rrbracket_s = v \quad \llbracket x \rrbracket_s = \begin{cases} s(x), & x \in \text{dom}(s) \\ \dagger, & \text{otherwise} \end{cases} \quad \llbracket E_1 + E_2 \rrbracket_s = \begin{cases} \llbracket E_1 \rrbracket_s + \llbracket E_2 \rrbracket_s, & \llbracket E_1 \rrbracket_s, \llbracket E_2 \rrbracket_s \in \text{Nat} \\ \dagger, & \text{otherwise} \end{cases}$$

The big-step operational semantics uses judgements of the form $\sigma, C \Downarrow_\gamma o : \sigma'$, read: given implementation context γ and starting from state σ , the execution of command C results in outcome $o \in O = \{ok, err, miss\}$ and state σ' . The outcome can either equal: *ok* (elided where possible), denoting a successful execution; *err*, denoting an execution faulting with a language error, or *miss*, denoting an execution faulting with a missing resource error.

► **Definition 5** (Operational Semantics). *The representative cases of the big-step operational semantics are given in Figure 2. The complete semantics is given in [24].*

The successful transitions are straightforward: for example, the *nondet* command generates an arbitrary natural number; the function call executes the function body in a store where the function parameters are given the values of the function arguments and the function locals are initialised to `null`; and the control flow statements behave as expected. Allocation requires the specified amount of contiguous cells (always available as heaps are finite), and lookup, mutation, and deallocation require the targeted cell not to have been freed.

The semantics stores error information in a dedicated program variable `err`, not available to the programmer. For simplicity of error messages, we assume to have a function `str : PExp → Str`, which serialises program expressions into strings. The faulting semantic transitions are split into *language errors*, which can be captured by program-logic reasoning, and *missing resource errors*, which cannot, as such errors break the frame property. Language errors arise due to, for example, expressions being incorrectly typed (e.g. `null + 1`) or an attempt to access deallocated cells (that is, the use-after-free error). On the other hand, missing resource errors arise from accessing cells that are not present in memory.

4 Exact Separation Logic

We introduce an exact separation logic for our programming language, giving the assertion language in §4.1, specifications in §4.2, and the program logic rules in §4.3.

4.1 Assertion Language

To define assertions and their meaning, we introduce *logical variables*, $x, y, z, \in \text{LVar}$, distinct from program variables, and define the set of *logical expressions* as follows:

$$E \in \text{LExp} \triangleq v \mid x \mid \times \mid E + E \mid E - E \mid \dots \mid E = E \mid \neg E \mid E \wedge E \mid \dots \mid E \cdot E \mid E : E \mid \dots$$

Note that we can use program expressions in assertions (for example, $E \in \text{Val}$), as they form a proper subset of logical expressions.

$$\begin{array}{c}
\frac{\llbracket E \rrbracket_s = v \quad s' = s[x \rightarrow v]}{(s, h), x := E \Downarrow_\gamma (s', h)} \quad \frac{n \in \mathbb{N} \quad s' = s[x \rightarrow n]}{(s, h), x := \text{nondet} \Downarrow_\gamma (s', h)} \quad \frac{\llbracket E \rrbracket_s = \text{false}}{(s, h), \text{while } (E) \text{ C} \Downarrow_\gamma (s, h)} \\
\\
\frac{\llbracket E \rrbracket_s = \text{true} \quad (s, h), \text{C} \Downarrow_\gamma \sigma'' \quad \sigma'', \text{while } (E) \text{ C} \Downarrow_\gamma o : \sigma'}{(s, h), \text{while } (E) \text{ C} \Downarrow_\gamma o : \sigma'} \quad \frac{f(\vec{x}) \{ \text{C}; \text{return } E' \} \in \gamma \quad \llbracket \vec{E} \rrbracket_s = \vec{v} \quad \text{pv}(\text{C}) \setminus \{ \vec{x} \} = \{ \vec{z} \} \quad s_p = \emptyset [\vec{x} \rightarrow \vec{v}] [\vec{z} \rightarrow \text{null}] \quad (s_p, h), \text{C} \Downarrow_\gamma (s_q, h') \quad \llbracket E' \rrbracket_{s_q} = v'}{(s, h), y := f(\vec{E}) \Downarrow_\gamma (s[y \rightarrow v'], h')} \\
\\
\frac{\llbracket E \rrbracket_s = n \quad h(n) = v}{(s, h), x := [E] \Downarrow_\gamma (s[x \rightarrow v], h)} \quad \frac{\llbracket E_1 \rrbracket_s = n \quad h(n) \in \text{Val} \quad \llbracket E_2 \rrbracket_s = v \quad h' = h[n \mapsto v]}{(s, h), [E_1] := E_2 \Downarrow_\gamma (s, h')} \\
\\
\frac{(n' + i \notin \text{dom}(h)) \Big|_{i=1}^{\llbracket E \rrbracket_s - 1} \quad h' = h \uplus \{ (n' + i \mapsto \text{null}) \Big|_{i=1}^{\llbracket E \rrbracket_s - 1} \}}{(s, h), x := \text{new}(E) \Downarrow_\gamma (s[x \rightarrow n'], h')} \quad \frac{\llbracket E \rrbracket_s = n \quad h(n) \in \text{Val}}{(s, h), \text{free}(E) \Downarrow_\gamma (s, h[n \mapsto \emptyset])} \\
\\
\frac{\llbracket E \rrbracket_s = \downarrow \quad v_{\text{err}} = [\text{"ExprEval"}, \text{str}(E)]}{(s, h), x := [E] \Downarrow_\gamma \text{err} : (s_{\text{err}}, h)} \quad \frac{\llbracket E \rrbracket_s = n \notin \text{dom}(h) \quad v_{\text{err}} = [\text{"MissingCell"}, \text{str}(E), n]}{(s, h), x := [E] \Downarrow_\gamma \text{miss} : (s_{\text{err}}, h)} \\
\\
\frac{h(\llbracket E \rrbracket_s) = \emptyset \quad v_{\text{err}} = [\text{"UseAfterFree"}, \text{str}(E_1), \llbracket E \rrbracket_s]}{(s, h), x := [E] \Downarrow_\gamma \text{err} : (s_{\text{err}}, h)} \quad \frac{\llbracket E \rrbracket_s = v \quad v_{\text{err}} = [\text{"Error"}, v]}{(s, h), \text{error}(E) \Downarrow_\gamma \text{err} : (s_{\text{err}}, h)}
\end{array}$$

■ **Figure 2** Operational semantics (excerpt), with $s_{\text{err}} \triangleq s[\text{err} \rightarrow v_{\text{err}}]$ and $\text{str} : \text{PExp} \rightarrow \text{Str}$.

► **Definition 6** (Assertion Language). *The assertion language is defined as follows:*

$$\begin{aligned}
\pi \in \text{BASrt} &\triangleq E_1 = E_2 \mid E_1 < E_2 \mid E \in X \mid \dots \mid \neg\pi \mid \pi_1 \Rightarrow \pi_2 \\
P \in \text{Asrt} &\triangleq \pi \mid \text{False} \mid P_1 \Rightarrow P_2 \mid \exists x. P \mid \text{emp} \mid E_1 \mapsto E_2 \mid E \mapsto \emptyset \mid P_1 \star P_2 \mid \otimes_{E_1 \leq x < E_2} P
\end{aligned}$$

where $E, E_1, E_2 \in \text{LExp}$, $X \subseteq \text{Val}$, and $x \in \text{LVar}$.

Boolean assertions, $\pi \in \text{BASrt}$, lift Boolean logical expressions to assertions. Assertions, $P \in \text{Asrt}$, contain Boolean assertions, standard first-order connectives and quantifiers, and spatial assertions. Spatial assertions include: the empty memory assertion, emp ; the positive cell assertion, $E_1 \mapsto E_2$; the negative cell assertion, $E \mapsto \emptyset$ (as in [11–14] and denoted in ISL by $E \not\mapsto$ [28]), the separating conjunction (star); and its iteration (iterated star).

To define assertion satisfiability, we introduce *substitutions*, $\theta : \text{LVar} \rightarrow_{\text{fin}} \text{Val}$, which are partial finite mappings from logical variables to values, extending expression evaluation of Definition 4 to $\llbracket E \rrbracket_{\theta, s}$ straightforwardly, with a new base case for logical variables:

$$\llbracket x \rrbracket_{\theta, s} = \theta(x), \text{ if } x \in \text{dom}(\theta) \quad \llbracket x \rrbracket_{\theta, s} = \downarrow, \text{ if } x \notin \text{dom}(\theta)$$

► **Definition 7** (Satisfiability). *The assertion satisfiability relation, denoted by $\theta, \sigma \models P$, is defined as follows:*

$$\begin{aligned}
\theta, (s, h) \models \pi &\Leftrightarrow \llbracket \pi \rrbracket_{\theta, s} = \text{true} \wedge h = \emptyset \\
\theta, (s, h) \models \text{False} &\Leftrightarrow \text{never} \\
\theta, (s, h) \models P_1 \Rightarrow P_2 &\Leftrightarrow \theta, (s, h) \models P_1 \Rightarrow \theta, (s, h) \models P_2 \\
\theta, (s, h) \models \exists x. P &\Leftrightarrow \exists v \in \text{Val}. \theta[x \mapsto v], (s, h) \models P \\
\theta, (s, h) \models \text{emp} &\Leftrightarrow h = \emptyset \\
\theta, (s, h) \models E_1 \mapsto E_2 &\Leftrightarrow h = \{ \llbracket E_1 \rrbracket_{\theta, s} \mapsto \llbracket E_2 \rrbracket_{\theta, s} \} \\
\theta, (s, h) \models E_1 \mapsto \emptyset &\Leftrightarrow h = \{ \llbracket E_1 \rrbracket_{\theta, s} \mapsto \emptyset \} \\
\theta, (s, h) \models P_1 \star P_2 &\Leftrightarrow \exists h_1, h_2. h = h_1 \uplus h_2 \wedge \theta, (s, h_1) \models P_1 \wedge \theta, (s, h_2) \models P_2 \\
\theta, (s, h) \models \otimes_{E_1 \leq x < E_2} P &\Leftrightarrow \exists h_i, \dots, h_{k-1}. h = \uplus_{j=i}^{k-1} h_j \wedge \forall j. i \leq j < k \Rightarrow \theta, (s, h_j) \models P[j/x] \\
&\text{where } i = \llbracket E_1 \rrbracket_{\theta, s}, k = \llbracket E_2 \rrbracket_{\theta, s}, \text{ and } x \text{ is not free in } E_1 \text{ or } E_2.
\end{aligned}$$

19:10 Exact Separation Logic

Assertion satisfiability is defined in the standard way. For convenience, we choose Boolean assertions to be satisfiable only in the empty heap.

► **Definition 8** (Validity). *An assertion P is valid, denoted by $\models P$, iff $\forall \theta, \sigma. \theta, \sigma \models P$.*

4.2 Specifications

We define specifications for commands and functions, focussing in particular on external and internal function specifications and the relationship between them.

► **Definition 9.** Specifications, $t = (P) (ok : Q_{ok}) (err : Q_{err}) \in \mathcal{Spec}$, comprise a pre-condition, P , a success post-condition, Q_{ok} , and a faulting post-condition, Q_{err} .

We denote that command C has specification t by $C : t$, or by $(P) C (ok : Q_{ok}) (err : Q_{err})$ in quadruple form. Additionally, we use the following shorthand:

$$\begin{aligned} (P) C (Q) &\triangleq (P) C (ok : Q) (err : \text{False}) \\ (P) C (err : Q) &\triangleq (P) C (ok : \text{False}) (err : Q) \\ (P) C (Q) &\triangleq (P) C (ok : -) (err : -) \end{aligned}$$

noting the use of Q for cases in which the post-condition details are not relevant. We use quadruples rather than triples since, even though the post-condition could be expressed as a disjunction of *ok*- and *err*-labelled assertions, we find the quadruple distinction helpful as compound commands (e.g. sequence) treat the two differently (cf. Figure 3).

The EX-validity of a specification t for a command C in an implementation context γ requires both OX and UX frame-preserving validity.

► **Definition 10** (γ -Valid Specifications). *Given implementation context γ , command C , and specification $t = (P) (ok : Q_{ok}) (err : Q_{err})$, t is γ -valid for C , denoted by $\gamma \models C : t$ or $\gamma \models (P) C (ok : Q_{ok}) (err : Q_{err})$, if and only if:*

$$\begin{aligned} & // \text{Frame-preserving over-approximating validity} \\ & (\forall \theta, s, h, h_f, o, s', h''. \theta, (s, h) \models P \implies \\ & \quad (s, h \uplus h_f), C \Downarrow_{\gamma} o : (s', h'') \implies (o \neq \text{miss} \wedge \exists h'. h'' = h' \uplus h_f \wedge \theta, (s', h') \models Q_o)) \wedge \\ & // \text{Frame-preserving under-approximating validity} \\ & (\forall \theta, s', h', h_f, o. \theta, (s', h') \models Q_o \implies h_f \# h' \implies \\ & \quad (\exists s, h. \theta, (s, h) \models P \wedge (s, h \uplus h_f), C \Downarrow_{\gamma} o : (s', h' \uplus h_f))) \end{aligned}$$

Observe that the outcome o can either be success or a language error; it cannot be a missing resource error as this would break UX frame preservation. As our operational semantics is complete, we can also use ESL to characterise non-termination. In particular, if a command satisfies a specification with both post-conditions **False**, then the execution of the command is guaranteed to not terminate if executed from a state satisfying the pre-condition. Were the semantics incomplete (for example, if it did not reason about errors), then such a specification might also indicate the absence of a semantic transition.

Compared to traditional OX reasoning, UX reasoning brings additional complexity to proofs of function specifications. To handle this complexity, we introduce two types of function specifications: *external* specifications, which provide the interface the function exposes to the client, and the related *internal* specifications, which provide the interface to the function implementation. This terminology is also used informally in InsecSL [25]. We use these in subsequent sections to show that ESL exhibits function compositionality.

► **Definition 11** (External Specifications). *A specification $(P) (ok : Q_{ok}) (err : Q_{err})$ is an external function specification if and only if:*

- $P = (\vec{x} = \vec{x} \star P')$, for some distinct program variables \vec{x} , distinct logical variables \vec{x} , and assertion P' , with $\text{pv}(P') = \emptyset$; and
- either $\text{pv}(Q_{ok}) = \{\text{ret}\}$ or $Q_{ok} = \text{False}$, and either $\text{pv}(Q_{err}) = \{\text{err}\}$ or $Q_{err} = \text{False}$.

The set of external specifications is denoted by \mathcal{ESpec} .

► **Definition 12** (Function Specification Contexts). *A function specification context, $\Gamma : \text{Fid} \rightarrow_{\text{fin}} \mathcal{P}(\mathcal{ESpec})$, is a finite partial function from function identifiers to a set of external specifications, with the more familiar notation $(\vec{x} = \vec{x} \star P) f(\vec{x}) (ok : Q_{ok}) (err : Q_{err}) \in \Gamma$ at times used in place of $(\vec{x} = \vec{x} \star P) (ok : Q_{ok}) (err : Q_{err}) \in \Gamma(f)$.*

The constraints on external specifications are well-known from OX logics and follow the usual scoping of function parameters and local variables, which are limited to the function body: the pre-conditions only contain the function parameters, \vec{x} ; and the post-conditions may only have the (dedicated) program variables ret or err , which hold, respectively, the return value on successful termination or the error value on faulting termination.

Internal function specifications are more interesting for exact and UX than for OX reasoning. The internal pre-condition is straightforward, extending the external pre-condition by instantiating the local variables to null . The internal post-condition must therefore include information about the parameters and local variables, as the internal specification cannot lose information. This means that the connection between internal and external post-conditions is subtle, given the constraints on the latter. To address this, we define an internalisation function, relating an external function specification with a set of possible internal specifications. In particular, the external post-condition has to be equivalent to an internal one in which the parameters and local variables of the internal post-condition have been replaced by fresh existentially quantified logical variables.

► **Definition 13** (Internalisation). *Given implementation context γ and function $f \in \text{dom}(\gamma)$, a function specification internalisation, $\text{Int}_{\gamma, f} : \mathcal{ESpec} \rightarrow \mathcal{P}(\text{Spec})$, is defined as follows:*

$$\begin{aligned} \text{Int}_{\gamma, f}((P) (ok : Q_{ok}) (err : Q_{err})) = \\ \{(P \star \vec{z} = \text{null}) (ok : Q'_{ok}) (err : Q'_{err}) \mid & \models Q'_{ok} \Rightarrow E \in \text{Val} \star \text{True} \text{ and} \\ & \models Q_{ok} \Leftrightarrow \exists \vec{p}. Q'_{ok}[\vec{p}/\vec{p}] \star \text{ret} = E[\vec{p}/\vec{p}] \text{ and} \\ & \models Q_{err} \Leftrightarrow \exists \vec{p}. Q'_{err}[\vec{p}/\vec{p}]\}, \end{aligned}$$

where $f(\vec{x})\{C; \text{return } E\} \in \gamma$, $\vec{z} = \text{pv}(C) \setminus \text{pv}(P)$, $\vec{p} = \text{pv}(P) \uplus \{\vec{z}\}$, and the logical variables \vec{p} are fresh with respect to Q_{ok} and Q_{err} .

This approach also works for SL and ISL as well (with \Leftarrow instead of \Leftrightarrow for the post-conditions for SL, and \Rightarrow instead of \Leftrightarrow for ISL). It is not strictly necessary for SL, however, as information about program variables can be forgotten in the internal post-conditions before the transition to the external post-condition.

► **Definition 14** (Environments). *An environment, (γ, Γ) , is a pair consisting of an implementation context γ and a specification context Γ .*

An environment (γ, Γ) is *valid* if and only if every function specified in Γ has an implementation in γ and every specification in Γ has a γ -valid internal specification.

► **Definition 15** (Valid Environments). *Given an implementation context γ and a specification context Γ , the environment (γ, Γ) is valid, written $\models (\gamma, \Gamma)$, if and only if*

$$\text{dom}(\Gamma) \subseteq \text{dom}(\gamma) \wedge (\forall f, \vec{x}, C, E. f(\vec{x})\{C; \text{return } E\} \in \gamma \implies (\forall t. t \in \Gamma(f) \implies \exists t' \in \text{Int}_{\gamma, f}(t). \gamma \models C : t'))$$

Finally, a specification t is valid for a command C in a specification context Γ if and only if t is γ -valid for all implementation contexts γ that validate Γ .

► **Definition 16** (Γ -Valid Specifications). *Given a specification context Γ , a command C , and a specification $t = (P) (Q)$, the specification t is Γ -valid for command C , written $\Gamma \models C : t$ or $\Gamma \models (P) C (Q)$, if and only if $\forall \gamma. \models (\gamma, \Gamma) \implies \gamma \models (P) C (Q)$.*

4.3 Program Logic

We give the representative ESL proof rules in Figure 3 and all in [24]. We introduce and discuss in detail the function-related rules, given for the first time in a UX setting. We denote the repetition of the pre-condition in the post-condition by *pre*. When reading the rules, it is important to remember that we *must not drop paths and must not lose information*. The judgement $\Gamma \vdash (P) C (ok : Q_{ok}) (err : Q_{err})$ means that the specification t is *derivable* for a command C given the specifications recorded in Γ .

The basic command rules are fairly straightforward. The [NONDET] rule existentially quantifies the generated value via $x \in \mathbb{N}$ to capture all paths, in contrast with the RHL [8] and ISL [28] rules, which explicitly choose one value to describe one path. The $E' \in \text{Val}$ in the post-condition is necessary as we know that E' evaluates to a value from the pre-condition and cannot lose information; the same principle applies to many other rules. The [ASSIGN] rule requires that the evaluation of E does not fault in the pre-condition via $E \in \text{Val}$. Strictly speaking, we should have an additional case in which the assigned variable is not in the store. To avoid this clutter, we instead assume that program variables are always in the store as we are analysing function bodies and, in our programming language, all local variables are initialised on function entry. The error-related rules capture cases in which expression evaluation faults (e.g. [LOOKUP-ERR-VAL] rule, using $E \notin \text{Val}$), expressions are of the incorrect type, or memory is accessed after it has been freed (e.g. [LOOKUP-ERR-USE-AFTER-FREE] rule, using $E \mapsto \emptyset$). Note that missing resource errors cannot be captured without breaking frame preservation, as the added-on frame could contain the missing resource.

When it comes to composite commands, we opt for two *if*-rules, covering the branches separately. The sequencing rule shows how exact quadruples of successive commands can be joined together, highlighting, in particular, how errors are collected using disjunction. One interesting aspect of this rule is what happens when C_1 only throws an error or does not terminate, meaning that $R = \text{False}$. In both those cases, given the exactness of the rules, it has to be that $Q_{ok} = Q_{err}^2 = \text{False}$, and the post-condition of the sequence becomes $(ok : \text{False}) (err : Q_{err}^1 \vee \text{False})$, meaning that, if C_1 only throws an error (that is, $Q_{err}^1 \neq \text{False}$) then that is the only error that can come out of the sequence, and if C_1 does not terminate (that is, $Q_{err}^1 = \text{False}$) then the sequence does not terminate either.

The while rule is an adaptation of the RHL while rule [8], generalising the invariant of the SL while rule with two natural-number-indexed families of variants, P_i and Q_i , which explicitly maintain the iteration index. Note how the i in the premise is a meta-variable representing a natural number, which in the conclusion gets substituted for an existentially quantified logical variable; a similar principle will be applied later when dealing with environment extension. Interestingly, this rule does not require adjustment to reason about non-termination.

$$\begin{array}{c}
\text{SKIP} \\
\Gamma \vdash (\text{emp}) \text{ skip } (\text{emp}) \\
\text{NONDET} \\
\frac{x \notin \text{pv}(E') \quad Q \triangleq E' \in \text{Val} \star x \in \mathbb{N}}{\Gamma \vdash (x = E') x := \text{nondet } (Q)} \\
\text{ASSIGN} \\
\frac{x \notin \text{pv}(E') \quad Q \triangleq E' \in \text{Val} \star x = E[E'/x]}{\Gamma \vdash (x = E' \star E \in \text{Val}) x := E (Q)} \\
\text{LOOKUP} \\
\frac{Q \triangleq E' \in \text{Val} \star x = E_1[E'/x] \star E[E'/x] \mapsto E_1[E'/x]}{\Gamma \vdash (x = E' \star E \mapsto E_1) x := [E] (Q)} \\
\text{MUTATE} \\
\frac{Q \triangleq E_1 \mapsto E_2 \star E \in \text{Val}}{\Gamma \vdash (E_1 \mapsto E \star E_2 \in \text{Val}) [E_1] := E_2 (Q)} \\
\text{NEW} \\
\frac{Q \triangleq E' \in \text{Val} \star \bigotimes_{0 \leq i < E[E'/x]} ((x+i) \mapsto \text{null})}{\Gamma \vdash (x = E' \star E \in \mathbb{N}) x := \text{new}(E) (ok : Q)} \\
\text{ERROR} \\
\frac{E_{\text{err}} \triangleq [\text{“Error”}, E]}{\Gamma \vdash (E \in \text{Val}) \text{ error}(E) (err : err = E_{\text{err}})} \\
\text{FREE} \\
\frac{Q \triangleq E' \in \text{Val} \star E \mapsto \emptyset}{\Gamma \vdash (E \mapsto E') \text{ free}(E) (ok : Q)} \\
\text{LOOKUP-ERR-VAL} \\
\frac{P \triangleq x = E' \star E \notin \text{Val} \quad E_{\text{err}} \triangleq [\text{“ExprEval”}, \text{str}(E)]}{\Gamma \vdash (P) x := [E] (err : Q_{\text{err}}^*)} \\
\text{LOOKUP-ERR-USE-AFTER-FREE} \\
\frac{P \triangleq x = E' \star E \mapsto \emptyset \quad E_{\text{err}} \triangleq [\text{“UseAfterFree”}, \text{str}(E), E]}{\Gamma \vdash (P) x := [E] (err : Q_{\text{err}}^*)} \\
\text{IF-THEN} \\
\frac{C \triangleq \text{if } (E) C_1 \text{ else } C_2 \quad \Gamma \vdash (P \star E) C_1 (Q)}{\Gamma \vdash (P \star E) C (Q)} \\
\text{IF-ELSE} \\
\frac{C \triangleq \text{if } (E) C_1 \text{ else } C_2 \quad \Gamma \vdash (P \star \neg E) C_2 (Q)}{\Gamma \vdash (P \star \neg E) C (Q)} \\
\text{IF-ERR-VAL} \\
\frac{C \triangleq \text{if } (E) C_1 \text{ else } C_2 \quad E_{\text{err}} \triangleq [\text{“ExprEval”}, \text{str}(E)]}{\Gamma \vdash (P \star E \notin \text{Val}) C (err : Q_{\text{err}}^*)} \\
\text{SEQ} \\
\frac{\Gamma \vdash (P) C_1 (ok : R) (err : Q_{\text{err}}^1) \quad \Gamma \vdash (R) C_2 (ok : Q_{\text{ok}}) (err : Q_{\text{err}}^2)}{\Gamma \vdash (P) C_1; C_2 (ok : Q_{\text{ok}}) (err : Q_{\text{err}}^1 \vee Q_{\text{err}}^2)} \\
\text{WHILE} \\
\frac{\forall i \in \mathbb{N}. \models P_i \Rightarrow E \in \mathbb{B} \star \text{True} \quad \forall i \in \mathbb{N}. \Gamma \vdash (P_i \star E) C (ok : P_{i+1}) (err : Q_i)}{\Gamma \vdash (P_0) \text{ while } (E) C (ok : \neg E \star \exists i. P_i) (err : \exists i. Q_i)} \\
\text{EQUIV} \\
\frac{\Gamma \vdash (P') C (ok : Q'_{\text{ok}}) (err : Q'_{\text{err}}) \quad \models P', Q'_{\text{ok}}, Q'_{\text{err}} \Leftrightarrow P, Q_{\text{ok}}, Q_{\text{err}}}{\Gamma \vdash (P) C (ok : Q_{\text{ok}}) (err : Q_{\text{err}})} \\
\text{FRAME} \\
\frac{\text{mod}(C) \cap \text{fv}(R) = \emptyset \quad \Gamma \vdash (P) C (ok : Q_{\text{ok}}) (err : Q_{\text{err}})}{\Gamma \vdash (P \star R) C (ok : Q_{\text{ok}} \star R) (err : Q_{\text{err}} \star R)} \\
\text{DISJ} \\
\frac{\Gamma \vdash (P_1) C (ok : Q_{\text{ok}}^1) (err : Q_{\text{err}}^1) \quad \Gamma \vdash (P_2) C (ok : Q_{\text{ok}}^2) (err : Q_{\text{err}}^2)}{\Gamma \vdash (P_1 \vee P_2) C (ok : Q_{\text{ok}}^1 \vee Q_{\text{ok}}^2) (err : Q_{\text{err}}^1 \vee Q_{\text{err}}^2)} \\
\text{EXISTS} \\
\frac{\Gamma \vdash (P) C (ok : Q_{\text{ok}}) (err : Q_{\text{err}})}{\Gamma \vdash (\exists x. P) C (ok : \exists x. Q_{\text{ok}}) (err : \exists x. Q_{\text{err}})}
\end{array}$$

■ **Figure 3** ESL proof rules (excerpt), with $Q_{\text{err}}^* = (\text{pre} \star \text{err} = E_{\text{err}})$.

The structural rules are not surprising, with equivalence replacing the forward/backward consequence of OX/UX reasoning and with frame, existential introduction, and disjunction affecting both post-conditions. Disjunction allows us to derive the standard SL if rule, which captures both branches at the same time. Note that there, however, is no sound conjunction rule, as the conjunction rules of SL and ISL cannot be combined in ESL, since conjunction does not distribute over the star in both directions, breaking frame preservation.

Function Call. We discuss the ESL function-call rule in detail, creating it starting from the standard OX-sound SL rule, adapted for quadruples:

$$\frac{\{\vec{x} = \vec{x} \star P\} f(\vec{x}) \{ok : Q_{\text{ok}}\} \{err : Q_{\text{err}}\} \in \Gamma \quad y \notin \text{pv}(E_y)}{\Gamma \vdash \{y = E_y \star \vec{E} = \vec{x} \star P\} y := f(\vec{E}) \{ok : Q_{\text{ok}}[y/\text{ret}]\} \{err : y = E_y \star Q_{\text{err}}\}}$$

In order to make this rule UX-sound, we only have to ensure that no information from the pre-condition is lost in the post-conditions. In particular, we have to remember:

19:14 Exact Separation Logic

- that the evaluation of E_y does not fault, captured by $E_y \in \text{Val}$ and needed in the success post-condition only, as it is already implied by the error post-condition; and
- that $\vec{E} = \vec{x}$ holds (with the substitution $[E_y/y]$ needed in the success post-condition as the value of y may change if the function call succeeds);

bringing us to the ESL function call rule:

$$\frac{(\vec{x} = \vec{x} \star P) \quad f(\vec{x}) \quad (ok : Q_{ok}) \quad (err : Q_{err}) \in \Gamma \quad y \notin \text{pv}(E_y) \quad Q'_{ok} \triangleq E_y \in \text{Val} \star \vec{E}[E_y/y] = \vec{x} \star Q_{ok}[y/\text{ret}] \quad Q'_{err} \triangleq y = E_y \star \vec{E} = \vec{x} \star Q_{err}}{\Gamma \vdash (y = E_y \star \vec{E} = \vec{x} \star P) \quad y := f(\vec{E}) \quad (ok : Q'_{ok}) \quad (err : Q'_{err})}$$

Environment Formation. Whereas the ESL function-call rule does not deviate substantially from its OX counterpart, the environment formation rules illustrate the difference in complexity between OX and UX function compositionality. These rules use the judgement $\vdash (\gamma, \Gamma)$ to state that the environment (γ, Γ) is *well-formed*. The base case, $\vdash (\emptyset, \emptyset)$, is trivial and the same as for SL, stating that the environment consisting of an empty implementation context and an empty specification context is well-formed. For illustrative purposes, we give a simplified version of the extension rule, extending the environment with a single, possibly recursive, function. The full rule, which extends the environment with a group of mutually recursive functions, is given in [24]. We start from the corresponding OX-sound rule from SL:

$$\frac{\text{ENV-EXTEND-SL} \quad \vdash (\gamma, \Gamma) \quad f \notin \text{dom}(\gamma) \quad \gamma' = \gamma[f \mapsto (\vec{x}, C, E)] \quad \Gamma' = \Gamma[f \mapsto \{t\}] \quad \exists t' \in \text{Int}_{\gamma', f}(t). \Gamma' \vdash C : t'}{\vdash (\gamma', \Gamma')}$$

which states that a well-formed environment (γ, Γ) can be extended with a given function f and its external specification t to (γ', Γ') if some corresponding internal specification of f can be proven for the body of f under the extended specification context Γ' .² Note that using Γ' means that a specification can be used to prove itself, which is sound in SL but unsound in ISL: specifically, it would allow us to prove UX-invalid specifications of non-terminating functions. For example, we would be able to prove that the function $\mathbf{f}() \{ r := \mathbf{f}(); \text{return } r \}$ satisfies the EX-valid specification **(emp)** $\mathbf{f}()$ **(False)**, but also the EX-invalid specification **(emp)** $\mathbf{f}()$ **(ret = 42)**. The latter is vacuously OX-valid as there are no terminating executions, but when considered from the UX viewpoint, it implies the existence of an execution path from the pre- to the post-condition, contradicting the non-termination of \mathbf{f} .

Therefore, to be soundly usable in UX reasoning, specifications with satisfiable post-conditions (onward: terminating specifications) must come with a mechanism that disallows the above counter-example. We achieve this by following a standard approach for reasoning about termination [7, 9], based on decreasing measures on well-ordered sets. In particular, we require the terminating specification to be proven, $t \triangleq (P) \quad (ok : Q_{ok}) \quad (err : Q_{err})$ to be extended with a *measure* $\alpha \in \mathbb{N}$, denoting this extension by $t(\alpha)$:³

$$t(\alpha) \triangleq (P \star \alpha = E_\mu) \quad (ok : Q_{ok} \star \alpha = E_\mu) \quad (err : Q_{err} \star \alpha = E_\mu)$$

where E_μ is a logical expression describing how the measure is computed. Then, we prove that $t(\alpha)$ holds for every specific α , assuming that recursive calls to f can only use the terminating

² Internalisation is normally omitted in SL as forward consequence allows information about program variables to be lost, making internal and external post-conditions of SL specifications almost the same.

³ We can extend the measure beyond natural numbers to computable ordinals, $\mathcal{O} \triangleq \omega_1^{\text{CK}}$, allowing us to reason about a broader set of functions, such as those with non-deterministic nested recursion (cf. [24]).

specifications $t(\beta)$ of a measure β *strictly smaller* than α . This restriction is standard and, if the proof succeeds, ensures that f has at least one terminating execution. Also, it disallows the above-mentioned counter-example, as no measure given in the pre-condition would be able to decrease before the recursive call. Importantly, the measure is only a tool required for proving specification validity and once this proof has been completed, the specification without the measure is added to Γ and can be used in proofs of client code.

In addition, we incorporate reasoning about non-terminating specifications (NT-specifications). This is relevant in situations in which the operational semantics of the analysed language is complete, which allows non-termination to be captured using the post-condition **False**. As NT-specifications are vacuously UX-sound, our focus is on ensuring their OX-soundness, which we do by again imposing a measure α , but allowing recursive calls for a measure β *smaller or equal* than α , that is, for an NT-specification to be used to prove itself. This, for example, allows for a proof of the specification **(emp)** $f() \{ r := f(); \text{return } r \}$ **(False)** by choosing a constant measure α . The ESL environment extension rule, therefore, is as follows:

```

ENV-EXTEND
// Assume valid environment, extend implementation context with new function f
⊢ (γ, Γ)    f ∉ dom(γ)    γ' = γ[f ↦ (x̄, C, E)]

// Extend the specifications of f with a measure α
t ≜ (P) (ok : Qok) (err : Qerr)    t∞ ≜ (P∞) (False)    t∞(α) ≜ (P∞ * α = Eμ) (False)
t(α) ≜ (P * α = Eμ) (ok : Qok * α = Eμ) (err : Qerr * α = Eμ)

// Construct Γ(α): assume t for measure β < α and t∞ for measure β ≤ α
Γ(α) = Γ[f ↦ {t(β) | β < α} ∪ {t∞ | β ≤ α}]

// For every α, prove internal specifications of f corresponding to t and t∞
∀α. ∃t' ∈ Intγ',f(t(α)). Γ(α) ⊢ C : t'    ∀α. ∃t' ∈ Intγ',f(t∞(α)). Γ(α) ⊢ C : t'

// Extend Γ with t and t∞
Γ' := Γ[f ↦ {t, t∞}]

```

$$\vdash (\gamma', \Gamma')$$

4.4 Soundness

We state the soundness results for ESL and give intuition about the proofs; the full proofs can be found in [24].

► **Theorem 17.** *Any derivable specification is valid:* $\Gamma \vdash (P) \text{ C } (Q) \implies \Gamma \models (P) \text{ C } (Q)$.

Proof. By induction on $\Gamma \vdash (P) \text{ C } (Q)$. Most cases are straightforward; the [FUN-CALL] rule obtains a valid specification for the function body from the validity of the environment. ◀

► **Theorem 18.** *Any well-formed environment is valid:* $\vdash (\gamma, \Gamma) \implies \models (\gamma, \Gamma)$.

Proof. At the core of the proof is a lemma stating that $\models (\gamma, \Gamma) \implies (\forall \alpha. \models (\gamma', \Gamma(\alpha)))$, where γ' and $\Gamma(\alpha)$ have been obtained from γ and Γ as per [ENV-EXTEND]. Using this lemma, we derive the desired $\models (\gamma', \Gamma')$, where Γ' is obtained from Γ and $\Gamma(\alpha)$ as per [ENV-EXTEND]. The proof of this lemma is done by transfinite induction on α and has the standard zero, successor, and limit ordinal cases. We outline the proof for the case in which a single, possibly recursive, function f with body C_f is added; the generalisation to n mutually recursive functions is straightforward and can be seen in [24].

In all three cases, the soundness of all specifications except the NT-specification with the highest considered ordinal follows from the inductive hypothesis. This remaining NT-specification is vacuously UX-valid, meaning that we only need to prove its OX-validity. For this, we use a form of fixpoint induction called Scott induction (see, e.g., Winskel [31]), required when specifications can be used to prove themselves (e.g. any SL specification).

We set up the Scott induction by extending the set of commands with two pseudo-commands, `scope` and `choice`, with the former modelling the function call but allowing arbitrary commands to be executed in place of the function body, and the latter denoting non-deterministic choice. We then construct the greatest-fixpoint closure of these extended commands, denoted by \mathbb{C} , whose elements may contain infinite applications of the command constructors. We define a behavioural equivalence relation $\simeq_{\gamma'}$ on \mathbb{C} and denote by $\mathbb{C}_{\gamma'}$ the obtained quotient space. This relation induces a partial order $\sqsubseteq_{\gamma'}$, and a join operator that coincides with `choice`, and we show that $(\mathbb{C}_{\gamma'}, \sqsubseteq_{\gamma'})$ is a domain.

We next define S^α as the set of all equivalence classes that hold an element that, for every specification in $(\Gamma(\alpha))(f)$, OX-satisfies at least one of its internal specifications, and show that S^α is an admissible subset of $\mathbb{C}_{\gamma'}$, that is, that it contains the least element of $\mathbb{C}_{\gamma'}$ (represented, for example, by the infinite loop `while (true) { skip }`) and is chain-closed.

We then define the function $h(C) \triangleq C_f[C, \gamma', f]$, which replaces all function calls to f in C_f with C using the `scope` command, and the function g as the lifting of h to $\mathbb{C}_{\gamma'}$: $g([C]) := [h(C)]$. We next prove that g is continuous (that is, monotonic and supremum-preserving) and that $g(S^\alpha) \subseteq S^\alpha$, from which we can apply the Scott induction principle, together with a well-known identity of the least-fixpoint, which implies that $C_f \in \text{lfp}(g)$, to obtain that $[C_f] \in S^\alpha$. From there, we are finally able to prove that $\models (\gamma', \Gamma(\alpha))$. ◀

These two theorems, to the best of our knowledge, are the first to demonstrate sound function compositionality for UX logics. Previous work on UX logics [25, 28, 29] used function specifications in examples but did not include rules in the logic for calling functions and managing a function specification environment. Program logics that do not include function rules and function specification environments in effect delegate soundness responsibilities to the meta-logic within which they are embedded. This might be appropriate in some contexts, such as in interactive theorem provers, whose meta-logic is reliable. Charguéraud’s clean-slate tutorial SL implementation in Coq [6], for example, does not provide either function call rules or program-logic-level infrastructure for a function specification environment; instead, it relies on Coq’s induction mechanism and definitional mechanism to use and store function specifications. However, when implementing an SL/ISL/ESL-based tool in a mainstream non-ITP language, such as C++ or OCaml, no reliable meta-logic that can act as a safety net is available. This is particularly concerning for UX logics, which require complex rules for handling functions, including forgetting information about function-local mutable variables. In ESL, the handling of functions is fully internalised into the logic, no meta-logic facilities are required to handle function calls, and the program-logic-level facilities of ESL for handling functions are validated by the soundness proof.

The proof of Theorem 18 can be adjusted for ISL: the function call rule would remain the same, and [ENV-EXTEND] would not include NT-specifications, removing the need for Scott induction. On the other hand, the Scott induction itself could be easily adapted for SL.

5 Examples: ESL in Practice

We demonstrate how to use ESL to specify and verify correctness and incorrectness properties of data-structure algorithms, focussing on singly-linked lists and binary search trees.

We investigate, for the first time, the use of abstract predicates in a UX program logic, decoupling abstraction from over-approximation. Our findings show that UX/EX specifications can soundly incorporate abstraction, but also that it, ultimately, cannot be used as freely as in the OX setting. Firstly, since UX reasoning cannot lose information, not all algorithms can be UX-specified at all levels of abstraction, and hence sometimes specifications have to be less abstract than in OX reasoning. Secondly, because specifications are only composable when expressed at the same level of abstraction, specifications of a library client have to be written at the “least common level of abstraction” of the specifications of all of the library functions that the client calls.

Building on §2, we give further intuition on how to think informally about UX/EX specifications using a number of list algorithms and predicates describing lists with various degrees of abstraction (§5.1, §5.2), more detail on how to write formal ESL proofs (§5.3), and examples of ESL reasoning for binary-search-tree algorithms (§5.4).

5.1 List Predicates

We implement singly-linked lists in the standard way: every list consists of two contiguous cells in the heap (denoted $x \mapsto a, b$, meaning $x \mapsto a \star x + 1 \mapsto b$), with the first holding the value of the node, the second holding a pointer to the next node in the list, and the list terminating with a `null` pointer. To capture lists in ESL, we use a number of list predicates:

$$\begin{aligned}
 \text{list}(x) &\triangleq (x = \text{null}) \vee (\exists v, x'. x \mapsto v, x' \star \text{list}(x')) \\
 \text{list}(x, n) &\triangleq (x = \text{null} \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star \text{list}(x', n - 1)) \\
 \text{list}(x, vs) &\triangleq (x = \text{null} \star vs = []) \vee (\exists v, x', vs'. x \mapsto v, x' \star \text{list}(x', vs') \star vs = v : vs') \\
 \text{list}(x, xs) &\triangleq (x = \text{null} \star xs = []) \vee (\exists v, x', xs'. x \mapsto v, x' \star \text{list}(x', xs') \star xs = x : xs') \\
 \text{list}(x, xs, vs) &\triangleq (x = \text{null} \star xs = [] \star vs = []) \vee \\
 &\quad (\exists v, x', xs', vs'. x \mapsto v, x' \star \text{list}(x', xs', vs') \star xs = x : xs' \star vs = v : vs')
 \end{aligned}$$

These predicates expose different parts of the list structure in their parameters, *hiding* the rest via existential quantification: the $\text{list}(x)$ predicate hides all information about the represented mathematical list, just declaring that there is a singly-linked list at address x ; the $\text{list}(x, n)$ predicate hides the internal node addresses and values, exposing the list length via the parameter n ; the $\text{list}(x, xs)$ predicate hides information about the values of the mathematical list, exposing the internal addresses of the list via the parameter xs ; the $\text{list}(x, vs)$ predicate hides information about the internal addresses, exposing the list’s values via the parameter vs ; and the list predicate $\text{list}(x, xs, vs)$ hides nothing, exposing the entire node-value structure via the parameters xs and vs . These predicates are related to each other via logical equivalence; for example, it holds that:

$$\begin{aligned}
 \models \text{list}(x) &\Leftrightarrow \exists n. \text{list}(x, n) & \models \text{list}(x) &\Leftrightarrow \exists vs. \text{list}(x, vs) \\
 \models \text{list}(x) &\Leftrightarrow \exists xs, vs. \text{list}(x, xs, vs) & \models \text{list}(x, n) &\Leftrightarrow \exists vs. \text{list}(x, vs) \star |vs| = n
 \end{aligned}$$

5.2 Writing UX/EX Abstract Specifications

We consider a number of list algorithms, described in words, and guide the reader on how to write correct UX/EX specifications for these algorithms using the list abstractions given in the previous section (§5.1), comparing how the UX/EX approach and specifications differ from their OX counterparts. We provide detailed proofs and implementations for each type of algorithm (iterative/recursive, allocating/deallocating, pure/mutative, etc.) in [24].

An important point is to understand how to look for counter-examples to a given specification: from the definition of OX validity, it follows that breaking OX reasoning amounts to “finding a state in the pre-condition (pre-model) for which the execution of f terminates and does not end in a state in the post-condition”; from the definition of UX validity, breaking UX reasoning means “finding a model of the post-condition (post-model) not reachable by execution of f from any state in the pre-condition”; and from the definition of EX validity, it follows that breaking EX reasoning means breaking either OX or UX reasoning. In addition, it is useful to remember that, for breaking UX validity, it is sufficient to find information known in the pre-condition but lost in the post-condition.

Length. We first revisit the list-length function $\text{LLen}(x)$, which takes a list at address x , does not modify it, and returns its length. In §2, we have shown that it satisfies the exact specification $(x = x \star \text{list}(x, n)) \text{LLen}(x) (\text{list}(x, n) \star \text{ret} = n)$ and observed that

(O1) abstraction does not always equal over-approximation.

Using similar reasoning, we can come to the conclusion that the following, less abstract specifications for $\text{LLen}(x)$ are also EX-valid:

$$\begin{aligned} (x = x \star \text{list}(x, vs)) \text{LLen}(x) (\text{list}(x, vs) \star \text{ret} = |vs|) \\ (x = x \star \text{list}(x, xs, vs)) \text{LLen}(x) (\text{list}(x, xs, vs) \star \text{ret} = |xs|) \end{aligned}$$

On the other hand, if we consider the following OX-valid specification:

$$\{x = x \star \text{list}(x)\} \text{LLen}(x) \{\exists n \in \mathbb{N}. \text{list}(x) \star \text{ret} = n\}$$

we see that it is not UX-valid as the post-condition does not connect the return value to the list. In particular, if we choose a post-model for $\text{list}(x)$ that has length 2, but then choose, for example, $\text{ret} = 42$, we run into a problem: as the algorithm does not modify the list, we have to choose the same model of $\text{list}(x)$ for the pre-model to have a chance of reaching the post-model, but then the algorithm will return 2, not 42, meaning that this specification is indeed not UX/EX-valid. From this discussion, we observe that:

(O2) in valid UX/EX specifications, data-structure abstractions used in a post-condition must expose enough information to capture the behaviour of the function being specified with respect to the information given in the pre-condition.

Note that, given a specification less abstract than strictly needed, one can obtain more abstract ones by using validity-preserving transformations on specifications that correspond to the structural rules of the logic. We refer to these as *admissible* transformations, give the ones for existential introduction and equivalence below, and the rest in [24]:

$$\text{ADM-EXISTS} \frac{\Gamma \models (\vec{x} = \vec{x} \star P) f(\vec{x}) (ok : Q_{ok}) (err : Q_{err}) \quad y \notin \vec{x}}{\Gamma \models (\vec{x} = \vec{x} \star \exists y. P) f(\vec{x}) (ok : \exists y. Q_{ok}) (err : err : \exists y. Q_{err})}$$

$$\text{ADM-EQUIV} \frac{\Gamma \models (\vec{x} = \vec{x} \star P') f(\vec{x}) (ok : Q'_{ok}) (err : Q'_{err}) \quad \models P', Q'_{ok}, Q'_{err} \Leftrightarrow P, Q_{ok}, Q_{err}}{\Gamma \models (\vec{x} = \vec{x} \star P) f(\vec{x}) (ok : Q_{ok}) (err : Q_{err})}$$

For list-length, starting from the least abstract specification using $\text{list}(x, xs, vs)$, we can derive, for example, the specification using $\text{list}(x, n)$, as follows:

$$\begin{aligned} & (x = x \star \text{list}(x, xs, vs)) \text{LLen}(x) (\text{list}(x, xs, vs) \star \text{ret} = |xs|) \\ [\text{ADM-EXISTS}] & (\exists vs. x = x \star \text{list}(x, xs, vs)) \text{LLen}(x) (\exists vs. \text{list}(x, xs, vs) \star \text{ret} = |xs|) \\ [\text{ADM-EXISTS}] & (\exists xs, vs. x = x \star \text{list}(x, xs, vs)) \text{LLen}(x) (\exists xs, vs. \text{list}(x, xs, vs) \star \text{ret} = |xs|) \\ [\text{ADM-EQUIV}] & (x = x \star \text{list}(x, n)) \text{LLen}(x) (\text{list}(x, n) \star \text{ret} = n) \end{aligned}$$

Interestingly, from one more application of [ADM-EXISTS] and [ADM-EQUIV], we can derive

$$(x = x \star \text{list}(x)) \text{ LLen}(x) (\exists n. \text{list}(x, n) \star \text{ret} = n)$$

which further illustrates observation (O2), in that even though the pre-condition does not talk about the length of the list, the post-condition has to expose it because the function output depends on it, and hence the post-condition must connect up the return value to the length of the list, here by an existentially quantified variable.

This approach of deriving abstract specifications can be used in general for working with ESL: for a given algorithm, first prove the least abstract specification, which exposes all details, and then adjust the degree of abstraction to fit the needs of the client code. We discuss this further in the upcoming paragraph on reasoning about client programs.

Membership. Next, we consider the list-membership function $\text{LMem}(x, v)$, which takes a list at address x , does not modify it, and returns `true` if v is in the list, and `false` otherwise. Given (O2) and the fact that the function output depends on the values in the list, we understand that, for its UX/EX specification, we should be using a list abstraction that exposes at least the values, that is, $\text{list}(x, vs)$ or $\text{list}(x, xs, vs)$. The corresponding specifications are:

$$\begin{aligned} (x = x \star v = v \star \text{list}(x, vs)) \text{ LMem}(x, v) (\text{list}(x, vs) \star \text{ret} = (v \in vs)) \\ (x = x \star v = v \star \text{list}(x, xs, vs)) \text{ LMem}(x, v) (\text{list}(x, xs, vs) \star \text{ret} = (v \in vs)) \end{aligned}$$

and are proven similarly to list-length. We can check that a more abstract specification, say:

$$\{x = x \star v = v \star \text{list}(x)\} \text{ LMem}(x, v) \{\exists b \in \mathbb{B}. \text{list}(x) \star \text{ret} = b\}$$

is not UX-valid, by choosing, as the post-model, b to be `false` and the list at x to contain v . As for list-length, since list-membership does not modify the list, we have to choose the same model of $\text{list}(x)$ for the pre-model to have a chance of reaching the post-model, but then the algorithm will return `true`, not `false`, so this post-model is not reachable.

Swap-First-Two. Next, we consider the list-swap-first-two function $\text{LSwapFirstTwo}(x)$, which takes a list at address x , swaps its first two values if the list is of sufficient length, returning `null`, and throws an error otherwise without modifying the list. Given (O2), to specify this function we need an abstraction that captures list length and, apparently, also the list values; for example, $\text{list}(x, vs)$. As this function can throw errors, its full EX specification has to use the ESL quadruple, in which the two post-conditions are constrained with the corresponding, shaded, branching conditions:

$$\begin{aligned} (x = x \star \text{list}(x, vs)) \\ \text{LSwapFirstTwo}(x, v) \\ (ok : \exists v_1, v_2, vs'. \text{vs} = v_1 : v_2 : vs' \star \text{list}(x, v_2 : v_1 : vs') \star \text{ret} = \text{null}) \\ (err : \text{list}(x, vs) \star |vs| < 2 \star err = \text{“List too short!”}) \end{aligned}$$

observing that the success post-condition, given the used abstraction, has to not only state that the length of the list is not less than two, but also how the values are manipulated, and also that the error message is chosen for illustrative purposes.

However, note that the swapped values *are not featured in the function output*, but instead remain contained within the predicate. This indicates that a more abstract specification:

$$\begin{aligned} (x = x \star \text{list}(x, n)) \\ \text{LSwapFirstTwo}(x, v) \\ (ok : \text{list}(x, n) \star n \geq 2 \star \text{ret} = \text{null}) (err : \text{list}(x, n) \star n < 2 \star err = \text{“List too short!”}) \end{aligned}$$

which only reveals the list length, might be EX-valid, and indeed it is. Any list we choose in the error post-model will have length less than two, and can then be used in the pre-model to reach the post-model. On the other hand, whichever list we choose in the success post-model will have length at least two, that is, its values will be of the form $v_1 : v_2 : vs$ and it will have some addresses, and then we can choose a list with the same addresses and values $v_2 : v_1 : vs$ in the pre-model and we will reach the post-model by executing the function.

Pointer-Reverse. Let us now examine the list-pointer-reverse function, $\text{LPRev}(x)$, which takes a list at address x and reverses it by reversing the direction of the next-pointers, returning the head of the reversed list. Given (O2) and the fact that the algorithm manipulates pointers and returns an address, but the actual values in the list are not exposed, we will try to use the address-only $\text{list}(x, xs)$ predicate to specify this function as in the following OX triple, where xs^\dagger denotes the reverse of the mathematical list xs :

$$\{x = x \star \text{list}(x, xs)\} \text{LPRev}(x) \{\text{list}(\text{ret}, xs^\dagger)\}$$

which would seem to be UX-valid given our OX experience and previous examples, but is not. In particular, it has no information about the logical variable x , which exists only in the pre-condition. This is not an issue in OX reasoning, but in UX reasoning it would mean that there exists a logical environment that interprets the post-condition but not the pre-condition, and such a specification, by the definition, could never be UX-valid.

To understand which specific information about x is required, we first add the general $x \in \text{Val}$, making the post-condition $\text{list}(\text{ret}, xs^\dagger) \star x \in \text{Val}$, and then try to choose a post-model by picking values for ret , xs , and x . Note that, given the definition of $\text{list}(x, xs)$, we cannot just pick any non-correlated values for ret and xs : in particular, either xs is an empty list and ret is `null`, or xs is non-empty and ret is its last element. This observation, in fact, reveals the information needed about x : either x is `null` and xs is empty, or xs is non-empty and x is its first element. We capture this information using the $\text{listHead}(x, xs)$ predicate:

$$\text{listHead}(x, xs) \triangleq (xs = [] \star x = \text{null}) \vee (\exists xs'. xs = x : xs')$$

and arrive at the desired EX specification of the list-pointer-reverse algorithm:

$$(x = x \star \text{list}(x, xs)) \text{LPRev}(x) (\text{list}(\text{ret}, xs^\dagger) \star \text{listHead}(x, xs))$$

Let us make sure that this specification is UX-valid. If we pick a post-model with $xs = []$, then $x = \text{ret} = \text{null}$ and the pre-model with the same x and xs will work, as the list holds no values. For a post-model with non-empty xs , x must equal the head of xs , ret must equal the tail of xs , and we also have to pick some arbitrary values vs , with $|vs| = |xs|$. Then, given the described behaviour of the algorithm, we know that this post-model is reachable from a pre-model which has the list at x with addresses xs and values vs^\dagger .

Free. Next, we take a look at the $\text{LFree}(x)$ function, which frees a given list at address x . Its OX specification is $\{x = x \star \text{list}(x)\} \text{LFree}(x) \{\text{ret} = \text{null}\}$, but it does not transfer to UX contexts because no resource from the pre-condition can be forgotten in the post-condition as that would break the UX frame property [28]. Instead, we have to keep track of the addresses to be freed, which we can do using the $\text{list}(x, xs)$ predicate (or $\text{list}(x, xs, vs)$), and we also have to explicitly state in the post-condition that these addresses have been freed:

$$(x = x \star \text{list}(x, xs)) \text{LFree}(x) (\text{freed}(xs) \star \text{listHead}(x, xs) \star \text{ret} = \text{null})$$

using the $\text{freed}(xs)$ predicate, which is defined as follows:

$$\text{freed}(xs) \triangleq (xs = []) \vee (\exists x, xs'. xs = x : xs' \star x \mapsto \emptyset, \emptyset \star \text{freed}(xs'))$$

Client Programs and Specification Composition. We discuss the usability of ESL specifications in general and abstraction in particular in the context of client programs that call multiple library functions. Consider the following (slightly contrived) client program, which takes a list and: pointer-reverses it if its length is between 5 and 10; frees it and then throws an error if its length is smaller than 5; and does not terminate otherwise:

```

LClient(x) {
  l := LLen(x);
  if (l < 5)
    { r := LFree(x); error("List too short!") } else
    { if (l > 10) { while (true) { skip } } else { r := LPRev(l) } };
  return r
}

```

Our first goal is to understand which is the most abstract list predicate that could be used for reasoning about this client, since we want to minimise the amount of details we need to carry along in the proof, noting that the least abstract one, $\text{list}(x, xs, vs)$, will always work. Observe that, importantly, only specifications expressed at the same abstraction level are composable with each other, because they must be composed using equivalence. We explore this in more detail in the subsequent formal discussion (see, in particular, observation (O5)).

When it comes to $\text{LClient}(x)$, for list-length, we need information about the list length, meaning that we can use either $\text{list}(x, n)$, $\text{list}(x, xs)$, or $\text{list}(x, vs)$, but not $\text{list}(x)$. For list-free, we must have information about the addresses, meaning that $\text{list}(x, n)$ and $\text{list}(x, vs)$ will not work, leaving us with $\text{list}(x, xs)$, which is also usable for list-pointer-reverse. Therefore, we can write the specification of this client using the $\text{list}(x, xs)$ predicate, as follows:

$$\begin{aligned}
& (x = x \star \text{list}(x, xs)) \\
& \quad \text{LClient}(x) \\
& \quad (ok : 5 \leq |xs| \leq 10 \star \text{list}(\text{ret}, xs^\dagger) \star \text{listHead}(x, xs)) \\
& \quad (err : |xs| < 5 \star \text{freed}(xs) \star \text{listHead}(x, xs) \star err = \text{"List too short!"})
\end{aligned}$$

In general, however, it is sufficient for a client to call one function that works with addresses and another that works with values for the only applicable predicate to be $\text{list}(x, xs, vs)$, which is still abstract in the sense that it allows for unbounded reasoning about lists, but does not hide any of its internal information. This leads us to the following observation:

(O3) specifications that use predicates which hide data-structure information, albeit provable, may have limited use in UX client reasoning.

As a final remark on abstraction, note that we have only considered predicates that expose the data-structure sub-parts (for lists, these sub-parts are values vs and addresses xs) either entirely or not at all. It would be also possible to expose *some* of this structure for some of the algorithms, but because of (O3), specifications using such abstractions are only composable with specifications exposing the same partial structure, and hence likely to be of limited use.

Non-termination. We conclude our discussion on specifications with two remarks on EX reasoning about non-terminating behaviour. First, consider the non-terminating branch of the LClient function, which is triggered when $|xs| > 10$. Observe that this branch is implicit in the client specification, in that it is subsumed by the success post-condition (since $\models P \vee (|xs| > 10 \star \text{False}) \Leftrightarrow P$). However, to demonstrate that it exists, we can constrain the pre-condition appropriately to prove the specification $(x = x \star \text{list}(x, xs) \star |xs| > 10) \text{LClient}(x) (\text{False})$. This implicit loss of non-terminating branches can be characterised informally as follows:

(O4) if the post-conditions do not cover all paths allowed by the pre-condition, then the “gap” is non-terminating.

In this case, the pre-condition implies $|xs| \in \mathbb{N}$ and the post-conditions cover the cases where $|xs| \leq 10$, leaving the gap when $|xs| > 10$, for which we provably have client non-termination.

Second, we observe that, in contrast to terminating behaviour, for non-terminating behaviour EX is as expressive as OX; that is, the EX triple $(P) C (\text{False})$ is equivalent to the OX triple $\{P\} C \{\text{False}\}$ as the UX triple $[P] C [\text{False}]$ is vacuously true. This is not to say that all non-terminating behaviour can be captured by ESL specifications. For example, as in OX, if the code branches on a value that does not come from the pre-condition, and if one of the resulting branches does not terminate, and if the code can also terminate successfully, then the non-terminating branch will be implicit in the pre-condition, but no gap in the sense of (O4) will be present. This is illustrated by the code and specification below, where the pre- and the post-condition are the same, but a non-terminating path still exists:

```
(x = 0) x := nondet; if (x > 42) { while (true) { skip } } else { x := 0 } (x = 0)
```

5.3 More ESL Proofs: Iterative list-length

In §2, we have shown a proof sketch for a recursive implementation of the list-length algorithm, demonstrating how to handle the measure for recursive function calls; how the folding of predicates works in the presence of equivalence; and how to move between external and internal specifications. We highlight again the UX-specific issue that we raised and that is related to predicate folding, which can be formulated generally as follows:

(O5) if the code accesses data-structure information that the used predicate hides, then that predicate might not be foldable in a UX-proof in all of the places in which it would be foldable in the corresponding OX-proof.

Here, we show how to write ESL proofs for looping code, using as example an iterative implementation of the list-length algorithm. Proofs for the majority of the other algorithms mentioned in §5.2 can be found in [24]; the rest are similar.

Iterative list-length in ESL: Proof Sketch. In Figure 4, we give an iterative implementation of the list-length algorithm and show that it satisfies the same ESL specification as its recursive counterpart, $(x = x \star \text{list}(x, n)) \text{LLen}(x) (ok : \text{list}(x, n) \star \text{ret} = n)$. Since there is no recursion, we elide the (trivial) measure. To state the loop variant, we use the list-segment predicate, defined as follows:

$$\text{lseg}(x, y, n) \triangleq (x = y \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star \text{lseg}(x', y, n - 1))$$

and to apply the [WHILE] rule, we define:

$$P_i \triangleq \exists j. \text{lseg}(x, x, i) \star \text{list}(x, j) \star n = i + j \star r = i$$

Note that we could have chosen to elide i from the body of P_i in this simple example, but since this is not necessarily possible or evident in general as well as for instructive purposes, we chose to keep it in the proof. Note how, on exiting the loop, the negation of the loop condition collapses the existentials i and j . This allows us to obtain the given internal post-condition, from which we then easily move to the desired external post-condition. For this proof, we also use three equivalence lemmas, which state that a non-empty list segment can be separated into its last element and the rest, that the length of an empty list equals zero, and that a null-terminated list-segment is a list.

```

 $\Gamma \vdash (x = x \star \text{list}(x, n))$ 
  LLen(x) {
    (  $x = x \star \text{list}(x, n) \star r = \text{null}$  )
    r := 0
    (  $x = x \star \text{list}(x, n) \star r = 0$  )
    (  $P_0$  )
    while (x  $\neq$  null) {
      (  $P_i \star x \neq \text{null}$  )
      (  $\exists j, v, x'. \text{lseg}(x, x, i) \star x \mapsto v, x' \star \text{list}(x', j - 1) \star n = i + j \star r = i$  )
      x := [x + 1];
      (  $\exists j, v, x'. \text{lseg}(x, x', i) \star x' \mapsto v, x \star \text{list}(x, j) \star n = i + (j + 1) \star r = i$  )
      // equivalence:  $\models \text{lseg}(x, y, n + 1) \Leftrightarrow \exists x', v. \text{lseg}(x, x', n) \star x' \mapsto v, y]$ 
      (  $\exists j. \text{lseg}(x, x, i + 1) \star \text{list}(x, j) \star n = (i + 1) + j \star r = i$  )
      r := r + 1
      (  $P_{i+1}$  )
    }
    (  $x = \text{null} \star \exists i. P_i$  )
    (  $\exists i, j. \text{lseg}(x, x, i) \star \text{list}(x, j) \star n = i + j \star r = i \star x = \text{null}$  )
    (  $\text{lseg}(x, \text{null}, n) \star r = n \star x = \text{null}$  ) // equivalence:  $\models \text{list}(\text{null}, j) \Leftrightarrow j = 0$ 
    (  $\text{list}(x, n) \star r = n \star x = \text{null}$  ) // equivalence:  $\models \text{lseg}(x, \text{null}, n) \Leftrightarrow \text{list}(x, n)$ 
    return r
    (  $\text{list}(x, n) \star r = n \star x = \text{null} \star \text{ret} = r$  )
    (  $\exists x_q, r_q. \text{list}(x, n) \star r_q = n \star x_q = \text{null} \star \text{ret} = r_q$  )
    (  $\text{list}(x, n) \star \text{ret} = n$  )
  }
  (  $\text{list}(x, n) \star \text{ret} = n$  )

```

■ **Figure 4** ESL proof sketch: iterative list-length.

5.4 Beyond List Examples: Binary Search Trees

While list algorithms illustrate many aspects of exact reasoning, it is also important to understand how ESL specification and verification works with other data structures. For this reason, we discuss two algorithms operating over binary search trees (BSTs) that are intended to represent sets of natural numbers. We use two abstractions for BSTs, one in which only their values are considered as a mathematical set:

$$\text{BST}(x, K) \triangleq (x = \text{null} \star K = \emptyset) \vee (\exists k, l, r, K_l, K_r. x \mapsto k, l, r \star \text{BST}(r, K_r) \star \text{BST}(l, K_l) \star K = K_l \uplus \{k\} \uplus K_r \star K_l < k \star k < K_r)$$

and another that fully exposes the BST structure:

$$\text{BST}(x, \tau) \triangleq (x = \text{null} \star \tau = \tau_\emptyset) \vee (\exists k, l, r, \tau_l, \tau_r. E \mapsto k, l, r \star \text{BST}(r, \tau_r) \star \text{BST}(l, \tau_l) \star \tau = ((x, k), \tau_l, \tau_r) \star \tau_l < k \star k < \tau_r)$$

where τ is a mathematical tree, that is, an algebraic data type with two constructors representing, respectively, an empty tree and a root node with two child trees: $\tau \in \text{Tree} \triangleq \tau_\emptyset \mid ((x, n), \tau_l, \tau_r)$, where the notation (x, n) represents a BST node with address x and value n . Note the overloaded $<$ notation, where one of the operands can be a set or a tree, which carry the intuitive meaning.

BST algorithms. We first consider the BST-find-minimum algorithm, $\text{BSTFindMin}(x)$, which takes a tree with root at x , does not modify it, and returns its minimum element or throws an empty-tree error. Since that algorithm operates only on the values in the tree, we are able to state its ESL specification using the $\text{BST}(x, K)$ predicate as follows:

$$(\mathbf{x} = x \star \text{BST}(x, K)) \text{BSTFindMin}(x) \begin{array}{l} (\text{ok}: x \neq \text{null} \star \text{BST}(x, K) \star \text{ret} = \min(K)) \\ (\text{err}: x = \text{null} \star \text{BST}(x, K) \star \text{err} = \text{“Empty tree!”}) \end{array}$$

We have also considered the BST-insert algorithm, $\text{BSTInsert}(x, v)$, which takes a tree with root at x and inserts a new node with value v into it as a leaf if v is not already in the tree, or leaves the tree unmodified if it is. As this algorithm interacts both with values and addresses in the tree, the appropriate abstraction for it is $\text{BST}(x, \tau)$, and its ESL specification is:

$$\begin{array}{c} (\mathbf{x} = x \star v = v \star \text{BST}(x, \tau)) \\ \text{BSTInsert}(x, v) \\ (\exists x'. \text{BST}(\text{ret}, \text{BSTInsert}(\tau, (x', v))) \star \text{BSTRoot}(x, \tau)) \end{array}$$

where $\text{BSTInsert}(\tau, \nu)$ is the mathematical algorithm that inserts the node ν into the tree τ :

$$\text{BSTInsert}(\tau_\emptyset, (x', v)) \triangleq \text{BSTInsert}(((x, k), \tau_l, \tau_r), (x', v)) \triangleq \begin{array}{l} \text{if } v < k \text{ then } ((x, k), \text{BSTInsert}(\tau_l, (x', v)), \tau_r) \\ \text{else if } k < v \text{ then } ((x, k), \tau_l, \text{BSTInsert}(\tau_r, (x', v))) \\ \text{else } ((x, k), \tau_l, \tau_r) \end{array}$$

and the predicate $\text{BSTRoot}(x, \tau)$ is defined analogously to $\text{listHead}(x, xs)$:

$$\text{BSTRoot}(x, \tau) \triangleq (\tau = \tau_\emptyset \star x = \text{null}) \vee (\exists k, \tau_l, \tau_r. \tau = ((x, k), \tau_l, \tau_r))$$

This example shows how in EX verification, just as in OX verification, we end up relating an imperative heap-manipulating algorithm to its mathematical/functional counterpart (cf. Appel [3] for a recent reiteration of this idea). The additional work required is that EX mathematical models must be more detailed: we are, yet again, not allowed to lose information. In particular, in OX verification we could relate $\text{BSTInsert}(x, v)$ to mathematical sets, but in EX verification we must relate our imperative implementation to tree models, including both values and pointers. Moreover, our mathematical model of the algorithm, $\text{BSTInsert}(\tau, (x', v))$, must insert elements in the same way as the imperative implementation, that is, in this case at the leaves of the tree. The proofs for both algorithms are given in [24].

6 Related Work

In the previous sections, we have placed ESL in the context of related work on OX and UX logics and associated tools. Here, we discuss formalisms capable of reasoning both about program correctness and program incorrectness, as well as existing approaches to the use of function specifications (summaries) and abstraction in symbolic execution.

Program Logics for Both Correctness and Incorrectness. Developed in parallel but independently of ESL, Outcome Logic (OL) [33], much like ESL, brings together reasoning about correctness and incorrectness into one logic. Both OL and ESL rely on the traditional meaning of correctness, but OL introduces a new approach to incorrectness, based on reachability of sets of states. It has not yet been shown that this approach has the same bug-finding potential as that of ISL: in particular, bi-abduction has not yet been demonstrated to be compatible with OL. In addition, the OL work, in contrast to ESL, does not discuss function compositionality or the interaction between abstraction, reachability, and incorrectness.

LCL_A [4,5] is a non-function-compositional, first-order logic that combines UX and OX reasoning using abstract interpretation. It is parametric on an abstract domain A , and proves UX triples of the form $\vdash_A [P] C [Q]$ where, under certain conditions, the triple also guarantees verification. These conditions, however, normally mean that only a limited number of pre-conditions can be handled. The conditions also have to be checked per-command and if they fail to hold (due to, e.g., issues with Boolean guards, which are known to be a major source of incompleteness), then the abstract domain has to be incrementally adjusted; the complexity of this adjustment and the expressivity of the resulting formalism is unclear.

Compositional Symbolic Execution. There exists a substantial body of work on symbolic execution with function summaries (e.g. [1, 15–17, 22, 32]), which is primarily based on first-order logic. We highlight the work of Godefroid et al., which initially used exact summaries of bounded program behaviour to drive the compositional dynamic test generation of SMART [15], and later distinguished between may (OX) and must (UX) summaries, leveraging the interaction between them to design the SMASH tool for compositional property checking and test generation [16]. SMASH, however, is limited in its ability to reason about heap-manipulating programs because, for example, it lacks support for pointer arithmetic. Nevertheless, it shows that interactions between OX and UX summaries can be exploited for automation, which is an important consideration for any automation of ESL. For example, SMASH is able to use not-may summaries (which amount to non-reachability) when constructing must-summaries (which amount to reachability), using the former to restrict the latter. When it comes to abstraction, for example, Anand et al. [2] implement linked-list and array abstractions for true bug-finding in non-compositional symbolic execution, in the context of the Java PathFinder, and use it to find bugs in list and array partitioning algorithms. True bug-finding is maintained by checking for state subsumption, which requires code modification rather than annotation and a record of all previously visited states.

7 Conclusions

We have introduced ESL, a program logic for exact reasoning about heap-manipulating programs. ESL specifications provide a sweet spot between verification and true bug-finding: as SL specifications, they capture all terminating behaviour, and, as ISL specifications, they describe only results and errors that are reachable. ESL specifications are therefore compatible with tools that target OX verification, such as VeriFast [19] and Iris [20], tools that target UX true bug-finding, such as Pulse [26, 28], and tools capable of targeting both, such as Gillian [10, 23]. ESL supports reasoning about mutually recursive functions and comes with a soundness result that immediately transfers to SL and ISL, thus demonstrating, for the first time, scalable functional compositionality for UX logics.

We have verified exact specifications for a number of illustrative examples, showing that ESL can reason about data-structure libraries, language errors, mutual recursion, and non-termination. In doing so, we emphasise the distinction between the often-conflated concepts of abstraction and over-approximation. We have demonstrated that abstract predicates can be soundly used in EX and UX reasoning, albeit not as freely as in OX reasoning.

We believe that ESL reasoning, in its intended context of semi-automatic verification of functional correctness properties, is useful for the verification of self-contained, critical code that underpins a larger codebase. To demonstrate this, we will in the future incorporate UX and EX verification inside Gillian [10, 23], which already has support for function compositionality and semi-automatic predicate management as part of its OX verification.

References

- 1 Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008. doi:10.1007/978-3-540-78800-3_28.
- 2 Saswat Anand, Corina S. Pasareanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1), 2009. doi:10.1007/s10009-008-0090-1.
- 3 Andrew W. Appel. Coq’s vibrant ecosystem for verification engineering. In *Conference on Certified Programs and Proofs (CPP)*, 2022. doi:10.1145/3497775.3503951.
- 4 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A logic for locally complete abstract interpretations. In *Symposium on Logic in Computer Science (LICS)*, 2021. doi:10.1109/LICS52264.2021.9470608.
- 5 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A correctness and incorrectness program logic. *Journal of the ACM*, 70(2), 2023. doi:10.1145/3582267.
- 6 Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages*, 4(ICFP), 2020. doi:10.1145/3408998.
- 7 Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *European Symposium on Programming (ESOP)*, 2016. doi:10.1007/978-3-662-49498-1_8.
- 8 Edsko de Vries and Vasileios Koutavas. Reverse Hoare logic. In *Software Engineering and Formal Methods (SEFM)*, 2011. doi:10.1007/978-3-642-24690-6_12.
- 9 Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19, 1967.
- 10 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part I: A multi-language platform for symbolic execution. In *Programming Language Design and Implementation (PLDI)*, 2020. doi:10.1145/3385412.3386014.
- 11 José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *Principles and Practice of Declarative Programming (PPDP)*, 2018. doi:10.1145/3236950.3236956.
- 12 José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: JavaScript verification toolchain. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158138.
- 13 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290379.
- 14 Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103656.2103663.
- 15 Patrice Godefroid. Compositional dynamic test generation. In *Principles of Programming Languages (POPL)*, 2007. doi:10.1145/1190216.1190226.
- 16 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Principles of Programming Languages (POPL)*, 2010. doi:10.1145/1706299.1706307.
- 17 Benjamin Hillery, Eric Mercer, Neha Rungta, and Suzette Person. Exact heap summaries for symbolic execution. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2016. doi:10.1007/978-3-662-49122-5_10.
- 18 C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM (CACM)*, 12(10), 1969. doi:10.1145/363235.363259.
- 19 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium (NFM)*, 2011. doi:10.1007/978-3-642-20398-5_4.

- 20 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2676726.2676980.
- 21 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O’Hearn. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), 2022. doi:10.1145/3527325.
- 22 Yude Lin, Tim Miller, and Harald Sondergaard. Compositional symbolic execution using fine-grained summaries. In *Australasian Software Engineering Conference*, 2015. doi:10.1109/ASWEC.2015.32.
- 23 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: Real-world verification for JavaScript and C. In *Computer Aided Verification (CAV)*, 2021. doi:10.1007/978-3-030-81688-9_38.
- 24 Petar Maksimović, Caroline Cronjäger, Andreas Löow, Julian Sutherland, and Philippa Gardner. Exact separation logic (extended version), 2023. arXiv:2208.07200.
- 25 Toby Murray, Pengbo Yan, and Gidon Ernst. Incremental vulnerability detection with insecurity separation logic, 2021. arXiv:2107.05225.
- 26 Peter W. O’Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2019. doi:10.1145/3371078.
- 27 Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, 2001. doi:10.1007/3-540-44802-0_1.
- 28 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O’Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided Verification (CAV)*, 2020. doi:10.1007/978-3-030-53291-8_14.
- 29 Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O’Hearn. Concurrent incorrectness separation logic. *Proceedings of the ACM on Programming Languages*, 6(POPL), 2022. doi:10.1145/3498695.
- 30 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002. doi:10.1109/LICS.2002.1029817.
- 31 Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction, Chapter 10*. MIT Press, 1993.
- 32 Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Principles of Programming Languages (POPL)*, 2008. doi:10.1145/1328438.1328467.
- 33 Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for correctness and incorrectness reasoning. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1), 2023. doi:10.1145/3586045.

Morpheus: Automated Safety Verification of Data-Dependent Parser Combinator Programs

Ashish Mishra  

Department of Computer Science, Purdue University, West Lafayette, IN, USA

Suresh Jagannathan  

Department of Computer Science, Purdue University, West Lafayette, IN, USA

Abstract

Parser combinators are a well-known mechanism used for the compositional construction of parsers, and have shown to be particularly useful in writing parsers for rich grammars with data-dependencies and global state. Verifying applications written using them, however, has proven to be challenging in large part because of the inherently effectful nature of the parsers being composed and the difficulty in reasoning about the arbitrarily rich data-dependent semantic actions that can be associated with parsing actions. In this paper, we address these challenges by defining a parser combinator framework called *Morpheus* equipped with abstractions for defining composable effects tailored for parsing and semantic actions, and a rich specification language used to define safety properties over the constituent parsers comprising a program. Even though its abstractions yield many of the same expressivity benefits as other parser combinator systems, *Morpheus* is carefully engineered to yield a substantially more tractable automated verification pathway. We demonstrate its utility in verifying a number of realistic, challenging parsing applications, including several cases that involve non-trivial data-dependent relations.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Parsers, Verification, Domain-specific languages, Functional programming, Refinement types, Type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.20

Related Version *Full Version*: <https://doi.org/10.48550/arXiv.2305.07901>

Funding Funding for this work was provided in part by DARPA under the SafeDocs program (grant HR0011-19-C-0073).

Acknowledgements The authors thank the reviewers for their insightful and useful comments.

1 Introduction

Parsers are transformers that decode serialized, unstructured data into a structured form. Although many parsing problems can be described using simple context-free grammars (CFGs), numerous real-world data formats (e.g., pdf [34], dns [9], zip [35], etc.), as well as many programming language grammars (e.g., Haskell, C, Idris, etc.) require their parser implementations to maintain additional context information during parsing. A particularly important class of context-sensitive parsers are those built from *data-dependent grammars*, such as the ones used in the data formats listed above. Such *data-dependent* parsers allow parsing actions that explicitly depend on earlier parsed data or semantic actions. Often, such parsers additionally use global effectful state to maintain and manipulate context information. To illustrate, consider the implementation of a popular class of *tag-length-data* parsers; these parsers can be used to parse image formats like PNG or PPM images, networking packets formats like TCP, etc., and use a parsed length value to govern the size of the input payload that should be parsed subsequently. The following BNF grammar captures this relation for a simplified PNG image.



© Ashish Mishra and Suresh Jagannathan;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 20; pp. 20:1–20:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```

png ::= header . chunk*
chunk ::= length . typespec . content

```

The grammar defines a `header` field followed by zero or more `chunks`, where each `chunk` has a single byte `length` field parsed as an unsigned integer, followed by a single byte `chunk type specifier`. This is followed by zero or more bytes of actual `content`. A useful data-dependent safety property that any parser implementation for this grammar should satisfy is that “*the length of content plus typespec is equal to the value of length*”.

Parser combinator libraries [44, 25, 12, 33] provide an elegant framework in which to write parsers that have such data-dependent features. These libraries simplify the task of writing parsers because they define the grammar of the input language and implement the recognizer for it at the same time. Moreover, since combinator libraries are typically defined in terms of a shallowly-embedded DSL in an expressive host language like Haskell [1, 19] or OCaml [25], parser implementations can seamlessly use a myriad of features available in the host language to express various kinds of data-dependent relations. This makes them capable of parsing both CFGs as well as richer grammars that have non-trivial semantic actions. Consequently, this style of parser construction has been adopted in many domains [2, 1, 33], a fact exemplified by their support in many widely-used languages like Haskell, Scala, OCaml, Java, etc.

Although parser combinators provide a way to easily write data-dependent parsers, verifying the correctness (i.e., ensuring that all data dependencies are enforced) of parser implementations written using them remains a challenging problem. This is in large part due to the inherently effectful nature of the parsers being composed, the pervasive use of rich higher-order abstractions available in the combinators used to build them, and the difficulty of reasoning about complex data-dependent semantic actions triggered by these combinators that can be associated with a parsing action.

This paper directly addresses these challenges. We do so by imposing modest constraints on the host language capabilities available to parser combinator programs; these constraints *enable* mostly automated reasoning and verification, *without* comprising the ability to specify parsers with rich effectful, data-dependent safety properties. We manifest these principles in the design of a deeply-embedded DSL for OCaml called **Morpheus** that we use to express and verify parsers and the combinators that compose them. Our design provides a novel (and, to the best of our knowledge, first) automated verification pathway for this important application class. This paper makes the following contributions:

1. It details the design of an OCaml DSL **Morpheus** that allows compositional construction of *data-dependent* parsers using a rich set of primitive parsing combinators along with an expressive specification language for describing safety properties relevant to parsing applications.
2. It presents an automated refinement type-based verification framework that validates the correctness of **Morpheus** programs with respect to their specifications and which supports fine-grained effect reasoning and inference to help reduce specification annotation burden.
3. It justifies its approach through a detailed evaluation study over a range of complex real-world parser applications that demonstrate the feasibility and effectiveness of the proposed methodology.

The remainder of the paper is organized as follows. The next section presents a detailed motivating example to illustrate the challenges with verifying parser combinator applications and presents a detailed overview of **Morpheus** that builds upon this example. We formalize **Morpheus**’s specification language and type system in Secs. 3 and 4. Details about **Morpheus**’s implementation and benchmarks demonstrate the utility of our framework is given in Sec. 5. Related work and conclusions are given in Secs. 6 and 7, respectively.

<pre> 1 decl ::= typedef . type-expr . id=rawident 2 extern ... 3 ... 4 typename ::= rawident 5 type-exp ::= "int" "bool" 6 expr ::= ... id=rawident </pre>	<pre> 1 decl ::= typedef . type-expr . id=rawident [2 ¬ id ∈ (!identifiers) 3 {types.add id} 4 ... 5 typename ::= x = rawident [x ∈ (!types)]{ 6 return x} 7 type-exp ::= "int" "bool" 8 expr ::= ... id=rawident {identifiers.add id ; 9 return id} </pre>
---	--

■ **Figure 1** Context-free and context-sensitive grammars for C declarations.

2 Motivation and Morpheus Overview

To motivate our ideas and give an overview of Morpheus, consider a parser for a simplified C language *declarations, expressions and typedefs* grammar. The grammar must handle context-sensitive disambiguation of *typename*s and *identifiers*¹. Traditionally, C-parsers achieve this disambiguation via cumbersome *lexer hacks*² which use feedback from the symbol table maintained in the parsing into the lexer to distinguish variables from types. Once the disambiguation is outsourced to the lexer-hack, the C-decl grammar can be defined using a context-free-grammar. For instance, the left hand side, Figure 1, presents a simplified context-free grammar production for a C declaration.

Unfortunately, ad-hoc lexer-hacks are both tedious and error prone. Further, this convoluted integration of the lexing and parsing phases makes it challenging to validate the correctness of the parser implementation. A cleaner way to implement such a parser is to disambiguate *typename*s and *identifiers* when parsing by writing an actual context-sensitive parser. One approach would be to define a shared *context* of two non-overlapping lists of *types* and *identifiers* and a stateful-parser using this context. The modified *context-sensitive* grammar is shown in right hand side, Figure 1.

The square brackets show context-sensitive checks e.g. $[\neg id \in (!identifier)]$ checks that the parsed *rawident* token *id* is not in the list of *identifiers*, while the braces show semantic actions associated with parser reductions, e.g. $\{typed.add id\}$, adds the token *id* to *types*, a list of *identifiers* seen thus far in the parse.

Given this grammar, we can use parser combinator libraries [25, 30] in our favorite language to implement a parser for C language declarations. Unfortunately, although cleaner than the using unwieldy lexer hacks, it is still not obvious how we might verify that implementations actually satisfy the desired *disambiguation* property, i.e. *typename*s and *identifiers* do not overlap. In the next section we provide an overview of Morpheus that informally presents our solution to this problem.

2.1 Morpheus Surface Language

An important design decision we make is to provide a surface syntax and API very similar to conventional monadic parser combinator libraries like Parsec [25] in Haskell or mParser [30] in OCaml; the core API that Morpheus provides has the signature shown in Figure 3. The library

¹ <https://web.archive.org/web/20070622120718/http://www.cs.utah.edu/research/projects/mso/goofie/grammar5.txt>

² <https://www.lysator.liu.se/c/ANSI-C-grammar-1.html>

20:4 Morpheus: Automated Safety Verification for Parser Combinators

```

1  let ids = ref []
2  let types = ref []
3  type decl =
4    Typeddecl of {typeexp:string}
5    | ...
6  type expression =
7    Address of expression
8    | Cast of string * expression
9    | ...
10   | Identifier of string
11
expression :
PEstexc
{∀ h,
 ldisjoint (sel (h, ids),sel (h, types)) = true}
ν : expression result
{∀ h, ν, h'.ν = Inl (v1) =>
 ldisjoint (sel (h', ids),sel (h', types)) = true}
∧ ν = Inr (Err) => included(inp,h,h') = true }

12 let expression =
13   dom char '('
14     tn ← typename
15     char ')'
16     e ← expression
17     return Cast (tn, e)
18   <|> ...
19   <|>
20   dom
21     id ← identifier
22     let b = List.mem id !types
23     if (!b) then
24       ids := id :: (!ids)
25       return (Identifier id)
26     else
27       fail

typeddecl :
PEstexc
{∀ h,
 ldisjoint (sel (h, ids),sel (h, types)) = true}
ν : tdecl result
{∀ h, ν, h'.ν = Inl (v1) =>
 ldisjoint (sel (h', ids),sel (h', types)) = true}
∧ ν = Inr (Err) => included(inp,h,h') = true }

let typeddecl =
  dom
    td ← keyword "typedef"
    te ← string "bool" <|> string "int"
    id ← identifier
    (* incorrect-check: if (not(List.mem id
    !types)) then*)
    if (not (List.mem id !ids)) then
      types := id :: (!types)
      return Tdecl {typeexp; id}
    else
      fail

typename :
PEstexc
{∀ h,
 ldisjoint (sel (h, ids),sel (h, types)) = true}
ν : string result
{∀ h, ν, h'.ν = Inl (v) =>
 mem (sel (h', types), v) = true
 ∧ ν = Inr (Err) => included(inp,h,h') = true}

let typename =
  dom
    x ← identifier
    if (List.mem x !types) then
      return x
    else
      fail

```

■ **Figure 2** A simplified C-declaration parser written in Morpheus. Specifications in blue are provided by the programmer; specifications in gray are inferred by Morpheus. Line number 28 represents the complete multiline type specification.

defines a number of primitive combinators: `eps` defines a parser for the empty language, `bot` always fails, and `char c` defines a parser for character `c`. Beyond these, the library also provides a `bind (>>=)` combinator for monadically composing parsers, a `choice (<|>)` combinator to non-deterministically choose among two parsers, and a `fix` combinator to implement recursive parsers. The `return x` is a parser which always succeeds with a value `x`. As we demonstrate, these combinators are sufficient to derive a number of other useful parsing actions such as `many`, `count`, etc. found in these popular combinator libraries. From the parser writer's perspective, Morpheus programs can be expressed using these combinators along with a basic collection of other non-parser expression forms similar to those found in an ML core language, e.g., first-class functions, `let` expressions, references, etc.

```

type 'a t
val eps : unit t
val bot : 'a t
val char : char → char t
val (>>=) : 'a t → (a → 'b t) → 'b t
val <|> : 'a t → 'a t → 'a t
val fix : ('b t → 'b t) → 'b t
val return : 'a → 'a t

```

■ **Figure 3** Signatures of primitive parser combinators supported by Morpheus.

For instance a parser for option p , which either parses an empty string or anything that p parses can be written:

```
let option p = (eps >>= λ_. return None) <|> (p >>= λ x. return Some x)
```

We can also define more intricate parsers like *Kleene-star* and *Kleene-plus*:

```
let star p = fix (λ p_star. eps <|> p >>= λ x. p_star >>= λ xs . return (x :: xs) )
let plus p = fix (λ p_star. p <|> p >>= λ x. p_star >>= λ xs . return (x :: xs) )
```

Figure 2 shows a Morpheus implementation that parses a valid C language decl.³ The parser uses two mutable lists to keep track of types and identifiers. The structure is similar to the original data-dependent grammar, even though the program uses ML-style operators for assignment and dereferencing. For ease of presentation, we have written the program using *do-notation* as syntactic sugar for Morpheus’s monadic bind combinator.

The `typedcl` parser follows the grammar and parses the keyword *typedef* using the keyword parser (not shown).⁴ It uses a choice combinator (`<|>`) (line 32), which has a semantics of a non-deterministic choice between two sub-parsers. The interesting case occurs while parsing an identifier (lines 33 - 39), in order to enforce disambiguation between *typenames* and *identifiers*, the parser needs to maintain an invariant that the two lists, `types` for parsed *typenames* and `ids` for parsed *identifiers* are always *disjoint* or *non-overlapping*.

In order to maintain the non-overlapping list invariant, a parsed identifier token (line 33) can be a valid typename only if it is not parsed earlier as an identifier expression. i.e. it is not in the `ids` list. The parser performs this check at (line 35). If this check succeeds, the list of typenames (`types`) is updated and a `decl` is returned, else the parsing fails.

The disambiguation decision is required during the parsing of an *expression*. The expression parser defines multiple choices. The parser for the *casting* expression parses a typename followed by a recursive call to expression. The `typename` parser in turn (line 41) parses an identifier token and checks that the identifier is indeed a `typename` (line 44) and returns it, or fails.

The `ids` list is updated during parsing an identifier expression (line 20), here again to maintain disambiguation, before adding a string to the `ids` list, its non-membership in the current `types` list is checked (line 22).

Although the above parser program is easy to comprehend given how closely it hews to the grammar definition, it is still nonetheless non-trivial to verify that the parser actually satisfies the required disambiguation safety property. For example, an implementation in

³ For now, ignore the specifications given in `gray` and `blue`.

⁴ Morpheus, like other parser combinator libraries provides a library of parsers for parsing keywords, identifiers, natural numbers, strings, etc.

which line 34 is replaced with the commented expression above it would incorrectly check membership on the wrong list. We describe how Morpheus facilitates verification of this program in the following section.

2.2 Specifying Data-dependent Parser Properties

Intuitively, verifying the above-given parser for the absence of overlap between the *typenames* and *identifiers* requires establishing the following partial correctness property: if the *types* and *identifiers* lists do not overlap when the *typedecl* parser is invoked, and the parser terminates without an error, then they must not overlap in the output state generated by the parser. Additionally, it is required that the parser consumes some prefix of the input list. Morpheus provides an expressive specification language to specify properties such as these.

Morpheus allows standard ML-style inductive type definitions that can be refined with *qualifiers* similar to other refinement type systems [38, 43, 18]. For instance, we can refine the type of a list of strings to only denote *non-empty* lists as: `type nonempty = { ν : [string] | len (ν) > 0 }`. Here, ν is a special bound variable representing a list and (`len ν > 0`) is a *refinement* where `len` is a *qualifier*, a predicate available to the type system that captures the length property of a list.

2.2.1 Specifying effectful safety properties

Standard refinement type systems, however, are ill-suited to specify safety properties for effectful computation of the kind expressible by parser combinators. Our specification language, therefore, also provides a type for effectful computations. We use a specification monad (called a *Parsing Expression*) of the form `PE ε { ϕ } ν : τ { ϕ' }` that is parameterized by the *effect* of the computation ε (e.g., `state`, `exc`, `nondet`, and their combinations like `stexc` for (both `state` and `exc`), `stnon` (for both `state` and `nondet`), etc.); and Hoare-style pre- and post-conditions [31, 41, 40]. Here, ϕ and ϕ' are first-order logical propositions over qualifiers applied to program variables and variables in the type context. The precondition ϕ is defined over an abstract input heap h while the postcondition ϕ' is defined over input heap h , output heap h' , and the special result variable ν that denotes the result of the computation. Using this monad, we can specify a safety property for the *typedecl* subparser as shown at line 28 in Figure 2. The type should be understood as follows: The *effect* label `stexc` defines that the parser may have both `state` effect as it reads and updates the context; and `exc` effect as the parser may fail. The precondition defines a property over a list of identifiers `ids` and a list of typenames `types` in the input heap h via the use of the built-in qualifier `sel` that defines a select operation on the heap [27]; here, ν is bound to the result of the parse. Morpheus also allows user-defined qualifiers, like the qualifier `ldisjoint`. It establishes the *disjointness/non-overlapping* property between two lists. This qualifier is defined using the following definition:

```

qualifier ldisjoint [] l2 → true
  | l1 [] → true
  | (x :: xs) l2 → member (x, l2) = false ∧ ldisjoint (xs, l2)
  | l1 (y :: ys) → member (y, l1) = false ∧ ldisjoint (l1, ys)

```

This definition also uses another qualifier for list membership called `member`. Morpheus automatically translates these user-defined qualifiers to axioms, logical sentences whose validity is assumed by the underlying theorem prover during verification. For instance, given the above qualifier, Morpheus generates axioms like:


```

Axiom1:  $\forall l1, l2 : \alpha \text{ list. } (\text{empty}(l1) \vee \text{empty}(l2)) \Rightarrow \text{ldisjoint}(l1, l2) = \text{true}$ 
Axiom2:  $\forall xs, l2 : \alpha \text{ list, } x : \alpha. \text{ldisjoint}(xs, l2) = \text{true} \wedge \text{member}(x, l2) = \text{false} \Rightarrow \text{ldisjoint}((x::xs), l2) = \text{true}$ 
Axiom3:  $\forall l1, l2 : \alpha \text{ ldisjoint}(l1, l2) \Leftrightarrow \text{ldisjoint}(l2, l1)$ 

```

The specification (at line 28) also uses another qualifier, `included(inp,h,h')`, which captures the monotonic consumption property of the input list `inp`. The qualifier is true when the remainder `inp` after parsing in `h'` is a suffix of the original `inp` list in `h`.

The types for other parsers in the figure can be specified as shown at lines 11, 40, etc.; these types shown in gray are automatically inferred by `Morpheus`'s type inference algorithm. For example, the type for the `typename` parser (line 40) returns an optional string (`result` is a special option type) and records that when parsing is successful, the returned string is added to the `types` list, and when unsuccessful, the input is still monotonically consumed.

2.2.2 Verification using Morpheus

Note that the pre-condition in the specification (`ldisjoint(l1, l2) = true`) and the type ascribed to the membership checks in the implementation (line 35) are sufficient to conclude that the addition of a `typename` to the `types` list (line 36) maintains the `ldisjoint` invariant as required by the postcondition.

In contrast, an erroneous implementation that omits the membership check or replaces the check at line 34 with the commented line above it will cause type-checking to fail. The program will be flagged ill-typed by `Morpheus`. For this example, `Morpheus` generated 21 verification conditions (VCs) for the control-path representing a successful parse and generated 5 VCs for the failing branch. We were able to discharge these VCs to the SMT solver Z3 [7], which took 6.78 seconds to verify the former and 1.90 seconds to verify the latter.

3 Morpheus Syntax and Semantics

3.1 Morpheus Syntax

Figure 4 defines the syntax of λ_{sp} , a core calculus for `Morpheus` programs. The language is a call-by-value polymorphic lambda-calculus with effects, extended with primitive expressions for common parser combinators and a refinement type-based specification language. A λ_{sp} value is either a constant drawn from a set of base types (`int`, `bool`, etc.), as well as a special `Err` value of type `exception`, an abstraction, or a constructor application. Variables bound to updateable locations (ℓ) are distinguished from variables introduced via function binding (x). A λ_{sp} expression e is either a value, an application of a function or type abstraction, operations to dereference and assign to top-level locations (see below), polymorphic **let** expressions, reference binding expressions, a **match** expression to pattern-match over type constructors, a **return** expression that lifts a value to an effect, and various parser primitive expressions that define parsers for the empty language (`eps`), a character (`char`) parser, and \perp , a parser that always fails. Additionally, the language provides combinators to monadically compose parsers ($>>=$), to implement parsers defined in terms of a non-deterministic choice of its constituents ($<|>$), and to express parsers that have recursive ($\mu(x : \tau).p$) structure.

We restrict how effects manifest by requiring reference creation to occur only within **let** expressions and not in any other expression context. Moreover, the variables bound to locations so created (ℓ) can only be dereferenced or assigned to and cannot be supplied

Expression Language

$c, \text{unit}, \text{Err}$	\in	<i>Constants</i>	
x	\in	<i>Vars</i>	
inp, ℓ	\in	<i>RefVars</i>	
v	\in	<i>Value</i>	$::= c \mid \lambda(x : \tau).e \mid \Lambda(\alpha).e \mid D_i \overline{t_k} \overline{v_j}$
e	\in	<i>Exp</i>	$::= v \mid x \mid p \mid e x \mid e[t] \mid \text{deref } \ell \mid \ell := e$ $\mid \text{let } x = v \text{ in } e \mid \text{let } \ell = \text{ref } e \text{ in } e$ $\mid \text{match } v \text{ with } D_i \overline{\alpha} \overline{x_j} \rightarrow e \mid \text{return } e$
p	\in	<i>Parsers</i>	$::= \text{eps} \mid \perp \mid \text{char } e \mid (\mu(x : \tau).p)$ $\mid p \gg e \mid p \langle \rangle p$

Specification Language

α	\in	<i>TypeVariables</i>	
TN	\in	<i>User Defined Types</i>	$::= \alpha \text{ list}, \alpha \text{ tree}, \dots$
t	\in	<i>Base Types</i>	$::= \alpha \mid \text{int} \mid \text{bool} \mid \text{unit} \mid \text{heap} \mid \text{TN} \mid t$ $\text{result} \mid t \text{ ref} \mid \text{exc}$
τ	\in	<i>Type</i>	$::= \{\nu : t \mid \phi\} \mid (x : \tau) \rightarrow \tau \mid \text{PE}^\varepsilon\{\phi_1\}\nu : t\{\phi_2\}$
ε	\in	<i>Effect Labels</i>	$::= \text{pure} \mid \text{state} \mid \text{exc} \mid \text{nondet} \mid \dots$
σ	\in	<i>Type Scheme</i>	$::= \tau \mid \forall \alpha. \tau$
Q	\in	<i>Qualifiers</i>	$::= \text{QualifierName}(\overline{x_i})$
ϕ, P	\in	<i>Propositions</i>	$::= \text{true} \mid \text{false} \mid Q \mid Q_1 = Q_2$ $\mid \neg \phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \forall(x : t).\phi$
Γ	\in	<i>Type Context</i>	$::= \emptyset \mid \Gamma, x : \sigma \mid \Gamma, \ell : \tau \text{ ref} \mid \Gamma, \phi$
Σ	\in	<i>Constructors</i>	$::= \emptyset \mid \Sigma, D_i \overline{\alpha_k} \overline{x_j} : \overline{\tau_j} \rightarrow \tau$

■ **Figure 4** λ_{sp} Expressions and Types.

as arguments to abstractions or returned as results since they are not treated as ordinary expressions. This stratification, while arguably restrictive in a general application context, is consistent with how parser applications, such as our introductory example are typically written and, as we demonstrate below, do not hinder our ability to write real-world data-dependent parser implementations. Enforcing these restrictions, however, provides a straightforward mechanism to prevent aliasing of effectful components during evaluation, significantly easing the development of an automated verification pathway in the presence of parser combinator-induced computational effects.

3.2 Semantics

Figure 5 presents a big-step operational semantics for λ_{sp} parser expressions; the semantics of other terms in the language is standard. The semantics is defined via an evaluation relation (\Downarrow) that is of the form $(\mathcal{H}; e) \Downarrow (\mathcal{H}'; v)$. The relation defines how a **Morpheus** expression e evaluates with respect to a heap \mathcal{H} , a store of locations to base-type values, to yield a value v , which can be a normal value or an exceptional one, the latter represented by the exception constant Err , and a new heap \mathcal{H}' .

$$\boxed{(\mathcal{H}; e) \Downarrow (\mathcal{H}'; v)}$$

$$\begin{array}{c}
\text{P-EPS} \frac{}{(\mathcal{H}; \text{eps}) \Downarrow (\mathcal{H}; ())} \quad \text{P-}\perp \frac{}{(\mathcal{H}; \perp) \Downarrow (\mathcal{H}; \text{Err})} \quad \text{P-FIX} \frac{(\mathcal{H}; [\mu x : \sigma.p/x]p) \Downarrow (\mathcal{H}'; v)}{(\mathcal{H}; \mu x : \sigma.p) \Downarrow (\mathcal{H}'; v)} \\
\\
\text{P-CHAR-TRUE} \frac{(\mathcal{H}; e) \Downarrow (\mathcal{H}; 'c') \quad \mathcal{H}(\text{inp}) = ('c' :: s) \quad \mathcal{H}' = \mathcal{H}[\text{inp} \mapsto s]}{(\mathcal{H}; \text{char } e) \Downarrow (\mathcal{H}'; 'c')} \\
\\
\text{P-CHAR-FALSE} \frac{(\mathcal{H}; e) \Downarrow (\mathcal{H}; 'c') \quad \mathcal{H}(\text{inp}) \neq ('c' :: s) \quad \mathcal{H}' = \mathcal{H}[\text{inp} \mapsto \text{inp}]}{(\mathcal{H}; \text{char } e) \Downarrow (\mathcal{H}'; \text{Err})} \\
\\
\text{P-BIND-SUCCESS} \frac{(\mathcal{H}; p) \Downarrow (\mathcal{H}'; v_1) \quad (\mathcal{H}'; e) \Downarrow (\mathcal{H}'; (\lambda x : \tau. e')) \quad (\mathcal{H}'; [v_1/x]e') \Downarrow (\mathcal{H}''; v_2)}{(\mathcal{H}; p \gg e) \Downarrow (\mathcal{H}''; v_2)} \\
\\
\text{P-BIND-ERR} \frac{(\mathcal{H}; p) \Downarrow (\mathcal{H}'; \text{Err})}{(\mathcal{H}; p \gg e) \Downarrow (\mathcal{H}'; \text{Err})} \\
\\
\text{P-CHOICE-L} \frac{(\mathcal{H}; p_1) \Downarrow (\mathcal{H}'; v_1)}{(\mathcal{H}; (p_1 <|> p_2)) \Downarrow (\mathcal{H}'; v_1)} \quad \text{P-CHOICE-R} \frac{(\mathcal{H}; p_2) \Downarrow (\mathcal{H}''; v_2)}{(\mathcal{H}; (p_1 <|> p_2)) \Downarrow (\mathcal{H}''; v_2)}
\end{array}$$

■ **Figure 5** Evaluation rules for λ_{sp} parser expressions.

The empty string parser (rule P-EPS) always succeeds, returning a value of type **unit**, without changing the heap. A “bottom” (\perp) parser on the other hand always fails, producing an exception value, also without changing the heap. If the argument e to a character parser **char** yields value (a char ‘c’), and ‘c’ is the head of the input string (denoted by **inp**) being parsed, the parse succeeds (rule P-CHAR-TRUE), consuming the input and returning ‘c’, otherwise, the parse fails, with the input not consumed and the distinguished **Err** value being returned (rule P-CHAR-FALSE). The fixpoint parser $\mu x.p$ (P-FIX) allows the construction of recursive parser expressions. The monadic bind parser primitive (rule P-BIND-SUCCESS) binds the result of evaluating its parser expression to the argument of the abstraction denoted by its second argument, returning the result of the evaluating the abstraction’s body (P-BIND-SUCCESS); the P-BIND-ERR rule deals with the case when the first expression fails. Evaluation of “choice” expressions, defined by rules P-CHOICE-L and P-CHOICE-R, introduce an unbiased choice semantics over two parsers allowing non-deterministic choices in parsers.

4 Typing λ_{sp} Expressions

4.1 Specification Language

The syntax of Morpheus’s type system is shown in the bottom of Figure 4 and permits the expression of *base types* such as integers, booleans, strings, etc., as well as a special **heap** type to denote the type of abstract heap variables like **h**, **h’** found in the specifications described below. There are additionally user-defined datatypes **TN** (**list**, **tree**, etc.), a special sum type (**t result**) to define two options of a successful and exceptional result respectively, and a

special exception type. More interestingly, base types can be refined with *propositions* to yield monomorphic refinement types. Such types [41, 38, 43] are either *base refinement types*, refining a base typed term with a refinement; *dependent function types*, in which arguments and return values of functions can be associated with types that are refined by propositions; or a *computation type* specifying a type for an effectful computation.

Effectful computations are refined using an effect specification monad:

$$\text{PE}^\varepsilon \{ \forall h. \phi_1 \} \nu : \mathbf{t} \{ \forall h, \nu, h'. \phi_2 \}$$

that encapsulates a base type \mathbf{t} , parameterized by an effect label ε , with Hoare-style pre- ($\{ \forall h. \phi_1 \}$) and post- ($\{ \forall h, \nu, h'. \phi_2 \}$) conditions. This type captures the behavior of a computation that (a) when executed in a pre-state with input heap h satisfies proposition ϕ_1 ; (b) upon termination, returns a value denoted by ν of base type \mathbf{t} along with output heap h' ; (c) satisfies a post-condition ϕ_2 that relates h , ν , and h' ; and (d) whose effect is over-approximated by effect label ε [20, 45]. An effect label ε is either (i) a **pure** effect that records an effect-free computation; (ii) a **state** effect that signifies a stateful computation over the program heap; (iii) an **exception** effect exc that denotes a computation that might trigger an exception; (iv) a **nondet** effect that records a computation that may have non-deterministic behavior; or (v) a *join* over these effects that reflect composite effectful actions. The need for the last is due to the fact that effectful computations are often defined in terms of a composition of effects, e.g. a parser oftentimes will define a computation that has a state effect along with a possible exception effect. To capture these composite effects, base effects can be joined to build a finite lattice that reflects the behavior of computations which perform multiple effectful actions, as we describe below.

Propositions (ϕ) are first-order predicate logic formulae over base-typed variables. Propositions also include a set of qualifiers which are applications of user-defined uninterpreted function symbols such as `mem`, `size` etc. used to encode properties of program objects, `sel` used to model accesses to the heap, and `dom` used to model membership of a location in the heap, etc. Proposition validity is checked by embedding them into a decidable logic that supports equality of uninterpreted functions and linear arithmetic (EUFLIA).

A type scheme (σ) is either a monotype (τ) or a universally quantified polymorphic type over type variables expressed in prenex-normal form ($\forall \alpha. \sigma$). A **Morpheus** specification is given as a type scheme.

There are two environments maintained by the **Morpheus** type-checker: (1) an environment Γ records the type of variables, which can include variables introduced by function abstraction as well as bindings to references introduced by `let` expressions, along with a set of propositions relevant to a specific context, and (2) an environment Σ maps datatype constructors to their signatures. Our typing judgments are defined with respect to a typing environment

$$\Gamma ::= . \mid \Gamma, x : \sigma \mid \Gamma, \ell : \tau \text{ ref}$$

that is either empty, or contains a list of bindings of variables to either type schemes or references. The rules have two judgment forms: $(\Gamma \vdash e : \sigma)$ gives a type for a **Morpheus** expression e in Γ ; and $(\Gamma \vdash \sigma_1 <: \sigma_2)$ defines a dependent subtyping rule under Γ .

Since our type expressions contain refinements, we generalize the usual notion of type substitution to reflect substitution within refined types:

$$\begin{aligned} [x_a/x] \{ \nu : \mathbf{t} \mid \phi \} &= \{ \nu : \mathbf{t} \mid [x_a/x] \phi \} \\ [x_a/x] (y : \tau) \rightarrow \tau' &= (y : [x_a/x] \tau) \rightarrow [x_a/x] \tau', y \neq x \\ [x_a/x] \text{PE}^\varepsilon \{ \phi_1 \} \{ \nu : \mathbf{t} \} \{ \phi_2 \} &= \text{PE}^\varepsilon \{ [x_a/x] \phi_1 \} \{ \nu : \mathbf{t} \} \{ [x_a/x] \phi_2 \} \end{aligned}$$

4.2 Typing Base Expressions

Figure 6 presents type rules for non-parser expressions. The type rules for non-reference variables, functions, and type abstractions (T-TYP-FUN) are standard. The syntax for function application restricts its argument to be a variable, allowing us to record the argument's (intermediate) effects in the typing environment when typing the application as a whole.

Base Expression Typing $\boxed{\Gamma \vdash e : \sigma}$

$$\begin{array}{c}
\text{T-VAR} \frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \quad \text{T-FUN} \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \rightarrow \tau_2} \quad \text{T-TYPAPP} \frac{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma}{\Gamma \vdash \Lambda\alpha.e[t] : [t/\alpha]\sigma} \\
\\
\text{T-APP} \frac{\Gamma \vdash e_f : (x : \{\nu : t \mid \phi_x\}) \rightarrow \text{PE}^\varepsilon\{\phi\} \quad \nu : t \{\phi'\} \quad \Gamma \vdash x_a : \{\nu : t \mid \phi_x\}}{\Gamma \vdash e_f x_a : [x_a/x]\text{PE}^\varepsilon\{\phi\} \quad \nu : t \{\phi'\}} \\
\\
\text{T-TYPFUN} \frac{\Gamma \vdash e : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\sigma} \quad \text{T-LET} \frac{\Gamma \vdash v : \forall\alpha.\sigma \quad \Gamma, x : \forall\alpha.\sigma \vdash e_2 : \sigma'}{\Gamma \vdash \text{let } x = v \text{ in } e_2 : \sigma'} \\
\\
\text{T-RETURN} \frac{\Gamma \vdash e : \{\nu : t \mid \phi\}}{\Gamma \vdash \text{return } e : \text{PE}^{\text{pure}}\{\forall h, \nu, h'. h' = h \wedge \phi\}} \\
\\
\text{T-CAPP} \frac{\Sigma(D_i) = \forall \overline{\alpha_k}. \overline{x_j} : \overline{\tau_j} \rightarrow \tau \quad \forall i, j. \Gamma \vdash v_j : [\overline{t_k}/\overline{\alpha_k}][\overline{v_j}/\overline{x_j}]\tau_j}{\Gamma \vdash D_i \overline{t_k} \overline{v_j} : [\overline{t}/\overline{\alpha}][\overline{v_j}/\overline{x_j}]\tau} \\
\\
\text{T-MATCH} \frac{\Sigma(D_i) = \forall \overline{\alpha_k}. \overline{x_j} : \overline{\tau_j} \rightarrow \tau_0 \quad \Gamma \vdash v : \tau_0 \quad \Gamma_i = \Gamma, \overline{\alpha_k}, \overline{x_j} : \overline{\tau_j} \quad \Gamma_i \vdash e_i : \text{PE}^\varepsilon\{\phi_i\} \quad \nu : t \{\phi_{i'}\}}{\Gamma \vdash \text{match } v \text{ with } D_i \overline{\alpha_k} \overline{x_j} \rightarrow e_i : \text{PE}^\varepsilon\{\forall h. \bigwedge_i (v = D_i \overline{\alpha_k} \overline{x_j}) \Rightarrow \phi_i\} \quad \nu : t \{\forall h, \nu', h'. \bigvee_i \phi_{i'}\}} \\
\\
\text{T-DEREF} \frac{\Gamma \vdash \ell : \text{PE}^{\text{state}}\{\phi_1\} \quad \nu : t \text{ ref } \{\phi_2\}}{\Gamma \vdash \text{deref } \ell : \text{PE}^{\text{state}}\{\forall h, \nu', h'. \text{dom}(h, \ell) = \nu' \wedge h = h'\}} \\
\\
\text{T-ASSIGN} \frac{\Gamma \vdash e : \{\nu : t \mid \phi\}}{\Gamma \vdash \ell := e : \text{PE}^{\text{state}}\{\forall h, \nu', h'. \text{sel}(h', \ell) = \nu' \wedge \phi(\nu')\}} \\
\\
\text{T-REF} \frac{\Gamma \vdash v : \{\nu : t \mid \phi\} \quad \Gamma, \ell : \text{PE}^{\text{state}}\{\forall h. \neg \text{dom}(h, \ell)\} \quad \nu' : t \text{ ref } \{\forall h, \nu', h'. \text{sel}(h', \ell) = v \wedge \phi(v) \wedge \text{dom}(h', \ell)\} \vdash e_b : \text{PE}^\varepsilon\{\text{dom}(h, \ell)\} \quad \nu'' : t \{\phi_b'\}}{\Gamma, h_i : \text{heap} \vdash \text{let } \ell = \text{ref } v \text{ in } e_b : \text{PE}^{\varepsilon \sqcup \text{state}}\{\forall h. \neg \text{dom}(h, \ell)\} \quad \nu'' : t \{\forall h, \nu'', h. \text{dom}(h_i, \ell) \wedge \text{sel}(h_i, \ell) = v \wedge \phi(v) \wedge \phi_b'\}}
\end{array}$$

■ **Figure 6** Typing Semantics for Morpheus Base Expressions.

The type rule for the return expression (T-RETURN) lifts its non-effectful expression argument e to have a computation effect with label **pure**, thereby allowing e 's value to be used in contexts where computational effects are required; a particularly important example of such contexts are bind expressions used to compose the effects of constituent parsers.

In the constructor application rule (T-CAPP), the expression’s type reflects the instantiation of the type and term variables in the constructor’s type with actual types and terms. A match expression is typed (rule T-MATCH) by typing each of the alternatives in a corresponding extended environment and returning a *unified type*. The pre-condition of the *unified* type is a conjunction of the pre-conditions for each alternative, while the post-condition over-approximates the behavior for each alternative by creating a disjunction of each of the possible alternative’s post-conditions. Location manipulating expressions (T-DEREF and T-ASSIGN) use qualifiers *sel* and *dom* to define constraints that reflect state changes on the underlying heap. The argument ℓ of a dereferencing expression (rule T-DEREF) is associated with a computation type over a *tref* base type. Its pre-condition requires ℓ to be in the domain of the input heap, and its post-condition establishes that ℓ ’s contents is the value returned by the expression and that the heap state does not change. The assignment rule (T-ASSIGN) assigns the contents of a top-level reference ℓ to the non-effectful value yielded by evaluating expression e . The pre-condition of its computation effect type requires that ℓ is in the domain of the input heap and that ℓ ’s contents in the output heap satisfies the refinement (ϕ) associated with its r-value. Finally, rule T-REF types a **let** expression that introduces a reference initialized to a value v . The body is typed in an environment in which ℓ is given a computational effect type. The pre-condition of this type requires that the input heap, i.e., the heap extant at the point when the binding of ℓ to *ref* v occurs, not include ℓ in its domain; its postcondition constrains ℓ ’s contents to be some value v' that satisfies the refinement ϕ associated with v , its initialization expression. The body of the **let** expression is then typed in this augmented type environment.

4.3 Typing Parser Expressions

Figure 7 presents the type rules for Morpheus parser expressions. The (T-SUB) rule defines the standard type subsumption rule. The empty string parser typing rule (T-P-EPS) assigns a type with *pure* effect and *unit* return type, while the postcondition establishes the equivalence of the input and the output heaps. The T-P-BOT rule captures the always failing semantics of \perp with an exception effect *exc* and corresponding return types and return values while maintaining the stability of the input heap. The type rules governing a character parser (T-P-CHAR) is more interesting because it captures the semantics of the success and the failure conditions of the parser. We use a sum type (α *result*) to define two options representing a successful and exceptional result, resp. (with the *Err* exception value in the latter case), using standard injection functions to differentiate among these alternatives. In the successful case, the returned value is equal to the consumed character, captured by an equality constraint over characters. In the successful case, the structure of the output heap with respect to the parse string *inp* must be the same as the input heap except for the absence of the 'c', the now consumed head-of-string character. In the failing case, the input remains unconsumed. Note that we also join the effect labels (*state* \sqcup *exc*), highlighting the state and exception effect. These effect labels form a standard join semi-lattice with an ordering relation (\leq)⁵.

Rule T-P-CHOICE defines the static semantics for a non-deterministic choice parser. It introduces a non-determinism effect to the parser’s composite type. The effect’s precondition requires that either of the choices can occur; we achieve this by restricting it to the conjunction of the two preconditions for the sub-parsers. The disjunctive post-condition requires that both the choices must imply the desired goal postcondition for a composite parser to be well-typed. The effect for the choice expression takes a join over the effects of the choices and the non-deterministic effect.

⁵ Details of the effect-labels and their join semi-lattice is provided in the accompanied technical report [28]

Parser Expression Typing $\boxed{\Gamma \vdash e : \sigma}$

$$\text{T-SUB} \frac{\Gamma \vdash e : \sigma_1 \quad \Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash e : \sigma_2}$$

$$\text{T-P-EPS} \frac{}{\Gamma \vdash \text{eps} : \text{PE}^{\text{pure}} \{\forall h. \text{true}\} \nu : \text{unit} \{\forall h, \nu, h'. h' = h\}}$$

$$\text{T-P-BOT} \frac{}{\Gamma \vdash \perp : \text{PE}^{\text{exc}} \{\forall h. \text{true}\} \nu : \text{exc} \{\forall h, \nu, h'. h' = h \wedge \nu = \text{Err}\}}$$

$$\text{T-P-CHAR} \frac{\begin{array}{c} \Gamma \vdash e : \{\nu' : \text{char} \mid \nu' = 'c'\} \\ \phi_2 = \forall h, \nu, h'. \forall x, y. \\ (\text{Inl}(x) = \nu \implies x = 'c' \wedge \text{upd}(h', h, \text{inp}, \text{tail}(\text{inp}))) \wedge \\ (\text{Inr}(y) = \nu \implies y = \text{Err} \wedge \text{sel}(h, \text{inp}) = \text{sel}(h', \text{inp})) \end{array}}{\Gamma \vdash \text{char } e : \text{PE}^{\text{state} \sqcup \text{exc}} \{\forall h. \text{true}\} \nu : \text{char result} \{\phi_2\}}$$

$$\text{T-P-CHOICE} \frac{\Gamma \vdash p_1 : \text{PE}^{\varepsilon} \{\phi_1\} \nu_1 : \tau \{\phi'_1\} \quad \Gamma \vdash p_2 : \text{PE}^{\varepsilon} \{\phi_2\} \nu_2 : \tau \{\phi'_2\}}{\Gamma \vdash (p_1 < | > p_2) : \text{PE}^{\varepsilon \sqcup \text{nondet}} \{(\phi_1 \wedge \phi_2)\} \nu : \tau \{(\phi'_1 \vee \phi'_2)\}}$$

$$\text{T-P-FIX} \frac{\Gamma, x : (\text{PE}^{\varepsilon} \{\phi\} \nu : \mathbf{t}\{\phi'\}) \vdash p : \text{PE}^{\varepsilon} \{\phi\} \nu : \mathbf{t}\{\phi'\} \quad x \notin FV(\phi, \phi')}{\Gamma \vdash \mu x : (\text{PE}^{\varepsilon} \{\phi\} \nu : \mathbf{t}\{\phi'\}). p : \text{PE}^{\varepsilon} \{\phi\} \nu : \mathbf{t}\{\phi'\}}$$

$$\text{T-P-BIND} \frac{\begin{array}{c} \Gamma \vdash p : \text{PE}^{\varepsilon} \{\phi_1\} \nu : \mathbf{t}\{\phi_{1'}\} \quad \Gamma \vdash e : (x : \tau) \rightarrow \text{PE}^{\varepsilon} \{\phi_2\} \nu' : \mathbf{t}'\{\phi_{2'}\} \\ \Gamma' = \Gamma, x : \tau, h_i : \text{heap} \quad h_i \text{ fresh} \\ \Gamma' \vdash p \gg e : \\ \text{PE}^{\varepsilon} \{\forall h. \phi_1 h \wedge \phi_{1'}(h, x, h_i) \Rightarrow \phi_2 h_i\} \\ \nu' : \mathbf{t}' \text{ result} \\ \{\forall h, \nu', h', y. (x \neq \text{Err} \Rightarrow \nu' = \text{Inl } y \wedge \phi_{1'}(h, x, h_i) \wedge \phi_{2'}(h_i, y, h')) \wedge \\ (x = \text{Err} \Rightarrow \nu' = \text{Inr } \text{Err} \wedge \phi_{1'}(h, x, h_i))\} \end{array}}{\Gamma \vdash p \gg e : \text{PE}^{\varepsilon} \{\forall h. \phi_1 h \wedge \phi_{1'}(h, x, h_i) \Rightarrow \phi_2 h_i\} \nu' : \mathbf{t}' \{\phi_{2'}\}}$$

Subtyping $\boxed{\Gamma \vdash \sigma_1 <: \sigma_2}$

$$\begin{array}{c} \text{T-SUB-BASE} \frac{\Gamma \vdash \{\nu : \mathbf{t} \mid \phi_1\} \quad \Gamma \vdash \{\nu : \mathbf{t} \mid \phi_2\}}{\Gamma \vDash \phi_1 \Rightarrow \phi_2} \quad \text{T-SUB-SCHEMA} \frac{\Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash \forall \alpha. \sigma_1 <: \forall \alpha. \sigma_2} \\ \text{T-SUB-ARROW} \frac{\Gamma \vdash \tau_{21} <: \tau_{11} \quad \Gamma \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash (x : \tau_{11}) \rightarrow \tau_{12} <: (x : \tau_{21}) \rightarrow \tau_{22}} \quad \text{T-SUB-TVAR} \frac{}{\Gamma \vdash \alpha <: \alpha} \\ \text{T-SUB-COMP} \frac{\Gamma \vDash \phi_2 \Rightarrow \phi_1 \quad \Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \varepsilon_1 \leq \varepsilon_2 \quad \Gamma, \phi_2 \vDash (\phi_{1'} \Rightarrow \phi_{2'})}{\Gamma \vdash \text{PE}^{\varepsilon_1} \{\phi_1\} \tau_1 \{\phi_{1'}\} <: \text{PE}^{\varepsilon_2} \{\phi_2\} \tau_2 \{\phi_{2'}\}} \end{array}$$

■ **Figure 7** Typing semantics for primitive parser expressions and subtyping rules.

Rule (T-P-FIX) defines the semantics for the terminating recursive fix-point combinator. Given an annotated type τ for the parameter x , if the type of the body p in an extended environment which has x mapping to τ , is τ , then τ is also a valid type for a recursive

fixpoint parser expression. The T-P-BIND rule defines a typing judgement for the exceptional monadic composition of a parser expression p with an abstraction e . The composite parser is typed in an extended environment (Γ') containing a binding for the abstraction's parameter x and an intermediate heap h_i that acts as the output/post-state heap for the first parser and the input/pre-state for the second. The relation between these heaps is captured by the inferred pre-and post-conditions for the composite parser. There are two possible scenarios depending upon whether the first parser p results in a success (i.e. $x \neq \text{Err}$) or a failure ($x = \text{Err}$).

In the successful case, the inferred conditions capture the following properties: a) the output of the combined parser is a success; b) the post-condition for the first expression over the intermediate heap h_i and the output variable x should imply the precondition of the second expression (required for the evaluation of the second expression); and, c) the overall post-condition relates the post-condition of the first with the precondition of the second using the intermediate heap h_i . The case when p fails causes the combined parser to fail as well, with the post-condition after the failure of the first as the overall post-condition. Note that the core calculus is sub-optimal in size since λ_{sp} supports both `return` and `eps`, even though the latter could be modeled using `return`. However, this design choice enables decidable typechecking by limiting the combination of higher-order functions, combinators and states. This is achieved using a limited bind $p \gg e$, rather than the general $e \gg e$, allowing for the definition of semantic actions e that only perform limited state manipulation, i.e., reading and updating locations. Thus \gg and $\langle | \rangle$ only take parser arguments; thus, $\text{eps} \langle | \rangle p$ is not equivalent to $(\text{return } () \langle | \rangle p)$, in fact the latter is disallowed. Another such design restriction shows up in the typing rules, e.g., the typing rule for function application (T-APP) restricts the arguments to be of *basetype*, thus disallowing expressions returning abstractions or computations, like $\text{return } (\lambda x. e)$ or $\text{return } (x := e)$. A more general definition for \gg will allow valid HO arguments, like $\lambda x. e \gg e1$, but translating such general HO stateful programs to decidable logic fragments is not always feasible, as is discussed in other fully dependent type systems [41].

The subtyping rules enable the propagation of refinement type information and relate the subtyping judgments to logical entailment. The subtyping rule for a base refinement (T-SUB-BASE) relates subtyping to the logical implication between the refinement of the subtype and the supertype. The (T-SUB-ARROW) rule defines subtyping between two function refinement types. The (T-SUB-COMP) rule for subtyping between computation types follows the standard Floyd-Hoare rule for *consequence*, coupled with the subtyping relation between result types and an ordering relation between effects (\leq). The subtyping rule for type variables (T-Sub-TVar) relates each type variable to itself in a reflexive way, while the subtyping for a type-schema lifts the subtyping relation from a schema to another schema.

4.4 Properties of the Type System

► **Definition 1** (Environment Entailment $\Gamma \models \phi$). Given $\Gamma = \dots, \overline{\phi_i}$, the entailment of a formula ϕ under Γ is defined as $(\bigwedge_i \phi_i) \implies \phi$

In the following, $\Gamma \models \phi(\mathcal{H})$ extends the notion of semantic entailment of a formula over an abstract heap $\Gamma \models \phi(h)$ to a concrete heap using an interpretation of concrete heap \mathcal{H} to an abstract heap h and the standard notion of well-typed *stores* ($\Gamma \vdash \mathcal{H}$).⁶

To prove soundness of Morpheus typing, we first state a soundness lemma for pure expressions (i.e. expressions with non-computation type).

⁶ Details are provided in the accompanied technical report [28].

► **Lemma 2** (Soundness Pure-terms). *If $\Gamma \vdash e : \{ \nu : \mathbf{t} \mid \phi \}$ then:*

- Either e is a value with $\Gamma \models \phi (e)$
- OR Given there exists a v , such that $(\mathcal{H}; e) \Downarrow (\mathcal{H}; v)$ then $\Gamma \vdash v : \mathbf{t}$ and $\Gamma \models \phi (v)$

► **Theorem 3** (Soundness Morpheus). *Given a specification $\sigma = \forall \bar{\alpha}. \text{PE}^\varepsilon \{ \phi_1 \} \nu : \mathbf{t} \{ \phi_2 \}$ and a Morpheus expression e , such that under some Γ , $\Gamma \vdash e : \sigma$, then if there exists \mathcal{H} such that $\Gamma \models \phi_1(\mathcal{H})$ then:*

1. Either e is a value, and: $\Gamma, \phi_1 \models \phi_2 (\mathcal{H}, e, \mathcal{H})$
2. Or, if there exists an \mathcal{H}' and v such that $(\mathcal{H}; e) \Downarrow (\mathcal{H}'; v)$, then
 - ◻ $\exists \Gamma', \Gamma \subseteq \Gamma'$ and (consistent $\Gamma \Gamma'$), such that:
 - a. $\Gamma' \vdash v : \mathbf{t}$.
 - b. $\Gamma', \phi_1 (\mathcal{H}) \models \phi_2 (\mathcal{H}, v, \mathcal{H}')$

where (consistent $\Gamma \Gamma'$) is a Boolean-valued function that ensures that $\forall x \in (\text{dom}(\Gamma) \cap \text{dom}(\Gamma'))$. $\Gamma \vdash x : \sigma \implies \Gamma' \vdash x : \sigma$. Additionally, $\forall \phi. \Gamma \models \phi \implies \Gamma' \models \phi$.

Proof. The soundness proof is by induction on typing rules in Figures 6 and 7, proving the soundness statement against the evaluation rules in Figures 5.⁷ ◀

5 Evaluation

5.1 Implementation

Morpheus is implemented as a deeply-embedded DSL in OCaml⁸ equipped with a refinement-type based verification system. It encodes the typing rules given in Section 4 and a parser translating an OCaml-based surface language of the kind presented in our motivating example to the Morpheus core, described in Section 3. To allow Morpheus programs to be easily used in an OCaml development, its specifications can be safely erased once the program has been type-checked. Note that a Morpheus program, verified against a safety specification is guaranteed to be safe when erased since verification takes place against a stricter memory abstraction; in particular, since Morpheus programs are free of aliasing by construction and thus remain so when evaluated as an ML program. This obviates the need for a separate interpreter/compilation phase and gives Morpheus-verified parsers efficiency comparable to the parsers written using OCaml parser-combinator libraries [30, 3].

Morpheus specifications typically require meaningful qualifiers over inductive data-types, beyond those discussed in our core language; in addition to the qualifiers discussed previously, typical examples include qualifiers to capture properties such as the length of a list, membership in a list, etc. Morpheus provides a way for users to write simple inductive propositions over inductive data types, translating them to axioms useful for the solver, in a manner similar to the use of *measures* and *predicates* in other refinement type works [38, 42]. For example, a qualifier for capturing the length property of a list can be written as:

```
qualifier len [] → 0 | len (x :: xs) → len (xs) + 1.
```

Morpheus generates the following axiom from this qualifier:

```
∀ xs : α list, x : α. len (x :: xs) = len (xs) + 1 ∧ len [] = 0
```

⁷ The **decidability** theorem and proofs for all the theorems are provided in the technical report [28].

⁸ <https://github.com/aegis-iisc/morpheus.git>

Morpheus is implemented in approximately 9K lines of OCaml code. The input to the verifier is a Morpheus program definition, correctness specifications, and any required qualifier definitions. Given this, Morpheus infers types for other expressions and component parsers, generates first-order verification conditions using the typing semantics discussed earlier, and checks the validity of these conditions.

5.2 Results and Discussions

We have implemented and verified the examples given in the paper, along with a set of benchmarks capturing interesting, real-world safety properties relevant to data-dependent parsing tasks. The goal of our evaluation is to consider the effectiveness of Morpheus with respect to generality, expressiveness and practicality. Table 1 shows a summary of the benchmark programs considered. Each benchmark is a Morpheus parser program affixed with a meaningful safety property (last column). The first column gives the name of the benchmark. The second column of the table describes benchmark size in terms of the number of lines of Morpheus code, without the specifications. The third column gives a pair D/P, showing the number of unique derived (D) combinators (like `count`, `many`, etc.) used in the benchmark from the Morpheus library, and the number of primitive (P) parsers (like `string`, `number`, etc.) from the Morpheus library used in the benchmark; the former provides some insight on the usability of our design choices in realizing extensibility. The fourth column lists the size of the grammar along with the number of production rules in the grammar. The fifth column gives the number of verification conditions generated, followed by the time taken to verify them (sixth column). The overall verification time is the time taken for generating verification conditions plus the time Z3 takes to solve these VCs. All examples were executed on a 2.7GHz, 64 bit Ubuntu. The next column quantifies the annotation effort for verification. It gives a ratio ($\#A/\#Q$) of required user-provided specifications (in terms of the number of conjuncts in the specification) to the total specification size (annotated + inferred). User-provided specifications are required to specify a top-level safety property and to specify invariants for fix expressions akin to loop invariants that would be provided in a typical verification task. Finally, the last column gives a high-level description of the data-dependent safety property being verified.

Our benchmarks explore data-dependent parsers from several interesting categories.⁹ The first category, represented by `ldris do-block`, `Haskell case-exp` and `Python while-statement`, capture parsing activities concerned with layout and indentation introduced earlier. Languages in which layout is used in the definition of their syntax require context-sensitive parser implementations [1, 2]. We encode a Morpheus parser for a sub-grammar for these languages whose specifications capture the layout-sensitivity property.

The second category, represented by `png` and `ppm` consider data-dependent image formats like PNG or PPM. Verifying data-dependence is non-trivial as it requires verifying an invariant over a monadic composition of the output of one parser component with that of a downstream parser component, interleaved with internal parsing logic.

The next category, captured by `xauction`, `xprotein`, and `health`, represent data-dependent parsing in data-processing pipelines over XML and CSV databases. For `xauction` and `xprotein`, we extend XPath expressions over XML to *dependent* XPath expressions. Given that XPath expressions are analogous to regular-expressions over structured XML data, *dependent* XPath expressions are analogous to dependent regular-expressions over XML.

⁹ The grammar for each of our implementations is given in the technical report [28].

■ **Table 1** Summary of Benchmarks : #Loc Loc defines the size of the parser implementation in Morpheus; D/P gives the number of derived/primitive combinator uses in the parser implementation; grammar size $G(\# \text{ prod})$ defines size of the grammar along with the number of production rules in the grammar; #VCs defines number of VCs generated; T(s) is the time for discharging these VCs in seconds; $(\#A/\#Q)$ defines the ratio of number of conjuncts used in the specification provided by the user ($\#A$) to the total number of conjuncts ($\#Q$) across all files in the implementation; Property gives a high-level description of the data-dependent safety property.

Name	# Loc	D/P	G(#prod)	# VCs	T (s)	(#A/#Q)	data-dependence
haskell	110	5/4	20 (7)	17	8.11	9/39	layout-sensitivity
idris	115	5/5	22(8)	33	10.46	7/26	layout-sensitivity
python	47	3/3	25 (7)	23	7.44	6/20	layout-sensitivity
ppm	46	5/2	21 (7)	20	5.33	4/9	tag-length-data
png chunk	30	3/4	10 (2)	12	3.38	2/7	tag-length-data
xauction	54	4/4	31 (10)	19	6.70	2/8	data-dependent XPath expression
xprotein	45	3/3	24(6)	22	6.23	4/10	data-dependent XPath expression
health	40	4/3	15(5)	13	4.56	2/8	data-dependent CSV pattern-matching
c typedef	60	4/4	14 (5)	21	6.78	4/16	context-sensitive dis- ambiguation
streams	51	4/2	12 (4)	16	5.21	2/9	safe stream manipu- lation

We use these expressions to encode a property of the XPath query over XML data for an online auction and protein database, resp. Note that verifying such properties over XPath queries is traditionally performed manually or through testing. In the case of **health**, we extend regular custom pattern-matching over CSV files to stateful custom pattern-matching, writing a data-dependent custom pattern matcher. We verify that the parser correctly checks relational properties between different columns in the database.

The next two categories have one example each: we introduced the **c typedef** parser in Section 2 that uses data dependence and effectful data structures to disambiguate syntactic categories (e.g., *typenamees* and *identifiers*) in a language definition. Benchmark **streams** defines a parser over streams (i.e. input list indexed with natural numbers).

5.2.1 Annotation overhead vs inference

There are some interesting things to note in the second to last column($\#A/\#Q$); First, as the benchmarks (grammars) become more complex, i.e., have a greater number of functions (sub-parsers), the ratio decreases (small is better). In other words, the gains of type-inference become more visible (e.g., **haskell**, **idris**, **c typedef**). The worst (highest) ratio is for the PPM parser. This parser is interesting because, even though the grammar is small, it makes multiple calls to fixpoint combinators. Thus, the user must provide specifications for the top-level parser and each fix-point combinator, thus increasing ($\#A$). Additionally, given a small number of functions (sub-parsers) due to small grammar size, the gains due to inference are also low. In summary, these trends show that the efforts needed for verification are at par with other Refinement typed languages like, Liquid Types [38], FStar [41], etc, and as the parsers become bigger, the benefits of inference become more prominent.

<pre> DoBlock ::= 'do' OpenBlock Do* CloseBlock; Do ::= 'let' Name TypeSig '=' Expr 'let' Expr '=' Expr Name '←' Expr Expr '←' Expr Ext Expr Expr </pre>	<pre> expr = do t ← term symbol "+" e ← expr pure t + e symbol '*' </pre>
--	---

(a) An Idris grammar rule for a do block.

(b) An input to the parser.

■ **Figure 8** An Idris grammar rule for a do block and an example input.

5.3 Case Study: Indentation Sensitive Parsers

As a case study to illustrate Morpheus’s capabilities, we consider a particular class of stateful parsers that are *indentation-sensitive*. These parsers are characterized by having indentation or layout as an essential part of their grammar. Because indentation sensitivity cannot be specified using a context-free grammar, their specification is often specified via an orthogonal set of rules, for example, the offside rule in Haskell.¹⁰ Haskell language specifications define these rules in a complex routine found in the lexing phase of the compiler [26]. Other indentation-sensitive languages like Idris [4] use parsers written using a parser combinator libraries like Parsec or its variants [25, 19] to enforce indentation constraints.

Consider the Idris grammar fragment shown in Figure 8a. The grammar defines the rule to parse a `do`-block. Such a block begins with the `do` keyword, and is followed by zero or more `do` statements that can be `let` expressions, a binding operation (`←`) over names and expressions, an external expression, etc. The Idris documentation specifies the indentation rule in English governing where these statements must appear, saying that the “*indentation of each do statement in a do-block Do* must be greater than the current indentation from which the rule is invoked* [13].” Thus, in the Idris code fragment shown in Figure 8b, indentation sensitivity constraints require that the last statement is not a part of the `do`-block, while the inner four statements are. A correct Idris parser must ensure that such indentation rules are preserved.

Figure 9 presents a fragment of the parser implementation in Haskell for the above grammar, taken from the Idris language implementation source, and simplified for ease of explanation. The implementation uses Haskell’s Parsec library, it implements indentation rules using a state abstraction (called `IState`) that stores the current indentation level as parsing proceeds. The parser then manually performs reads and updates to this state and performs indentation checks at appropriate points in the code (e.g. line 24, 53). The `IdrisParser` (line 8) is defined in terms of Parsec’s parser monad over an Idris state (here, `IState`), which along with other fields has an integer field (`ist`) storing the current indentation value. A typical indentation check (e.g. see lines 22 - 24) fetches the current value of `ist` using `getIst`, fetches the indentation of the next lexeme using the Parsec library function `indent`, and compares these values.

The structure of the implementation follows the grammar (Figure 8a): the `doBlock` parser parses a reserved keyword “`do`” followed by a block of `do_` statement lists. The indentation is enforced using the parser `indentedDoBlock` (defined at line 49) that gets the

¹⁰<https://www.haskell.org/onlinereport/haskell2010/haskellch10.html>

```

1  data IState = IState {
2    ist :: Int
3    ...
4  } deriving (Show)
5  data PTerm = PDoBlock [PDo]
6  data PDo t = DoExp t | DoExt t
7             | DoLet t t | ...
8  type IdrisParser a = Parser IState a
9
10 getIst :: IdrisParser IState
11 getIst = get
12 putIst :: (i : Int) → IdrisParser ()
13 putIst i = put {ist = i}
14
15 doBlock :: IdrisParser PTerm
16 doBlock = do
17     reserved "do"
18     ds ← indentedDoBlock
19     return (PDoBlock ds)
20 indentedDo :: IdrisParser (PDo PTerm)
21 indentedDo = do
22     allowed ← ist getIst
23     i ← indent
24     if (i <= allowed)
25         then fail ("end of block")
26         else do_
27 indent :: IdrisParser Int
28 indent =
29     do
30     if (lookAheadMatches (operator)) then
31         do
32             operator
33             return (sourceColumn.getSourcePos)
34     else
35         return (sourceColumn.getSourcePos)
36 do_ :: IdrisParser (PDo PTerm)
37 do_ = do
38     reserved "let"
39     i ← name
40     reservedOp "="
41     e ← expr
42     return (DoLet i e)
43 <|> do
44     e ← expr
45     return (DoExt i e)
46 <|> do e ← expr
47     return (DoExp e)
48 indentedDoBlock :: IdrisParser [PDo PTerm]
49 indentedDoBlock =
50     do
51         allowed ← ist getIst
52         lvl' ← indent
53         if (lvl' > allowed) then
54             do
55                 putIst lvl'
56                 res ← many (indentedDo)
57                 putIst allowed
58                 return res
59             else fail "Indentation error"
60
61 lookAheadMatches :: IdrisParser a →
62 IdrisParser Bool
63 lookAheadMatches p =
64     do
65         match ← lookAhead (optional p)
66         return (isJust match)

```

■ **Figure 9** A fragment of a Parsec implementation for Idris `do`-blocks with indentation checks.

current indentation value (`allowed`) and the indentation for the next lexeme using `indent`, checks that the indentation is greater than the current indentation (line 53) and updates the current indentation so that each `do` statement is indented with respect to this new value. It then calls a parser combinator `many` (line 56), which is the Parsec combinator for the Kleene-star operation, over the result of `indentedDo`, i.e., `indentedDo*`. The `indentedDo` parser again performs a manual indentation check, comparing the indentation value for the next lexeme against the block-start indentation (set earlier by `indentedDoBlock` at line 55) and, if successful, runs the actual `do_` parser (line 26). Finally, `indentedDoBlock` resets the indentation value to the value before the block (line 57).

Unfortunately, it is non-trivial to reason that these manual checks suffice to enforce the indentation sensitivity property we desire. Since they are sprinkled throughout the implementation, it is easy to imagine missing or misplacing a check, causing the parser to misbehave. More significantly, the implementation make incorrect assumptions about the

```

1  expr = do
2      t ← term
3      do
4          symbol "+"
5          e ← expr
6          pure t + e
7      "mplus" pure t

```

■ **Figure 10** An input expression that is incorrectly parsed by the implementation shown in Figure 9.

effective actions performed by the library that are reflected in API signatures. In fact, the logic in the above code has a subtle bug [1] that manifests in the input example shown in Figure 10.

Note that the indentation of the token “mplus” is such that it is not a part of either do block; the implementation, however, parses the last statement as a part of the inner do-block, thereby violating the indentation rule, leading to the program being incorrectly parsed.

The problem lies in a mismatch between the contract provided by the library’s `indent` function and the assumptions made about its behavior at the check at line 24 in the `indentedDo` parser (or similarly at line 53). Since checking indentation levels for each character is costly, `indent` is implemented (line 28) in a way that causes certain lexemes (user defined operators like “mplus”) to be ignored during the process of computing the next indentation level. It uses a `lookAheadMatches` parser to skip all lexemes that are defined as operators. In this example, `indent` does not check the indentation of lexeme “mplus”, returning the indentation of the token `pure` instead. Thus, the indentation of the last statement is considered to start at `pure`, which incorrectly satisfies the checks at line 24 or line 53, and thus causes this statement to be accepted as part of `indentedDoBlock`. Unearthing and preventing such bugs is challenging. We show how implementing the same parser in `Morpheus` allows us to catch the bug and verify a correct version of the parser. Figure 11 shows a `Morpheus` implementation for a portion of the Idris `doBlock` parser from Figure 9 showing the implementation of three parsers for brevity, `doBlock`, `indentedDo`, and `indent`, along with other helper functions. The structure is similar to the original Haskell implementation.

To specify an *indentation-sensitivity* safety property, we first define an inductive type for a parse-tree (`tree`) and refine this type using a dependent function type, (`offsideTree i`), that specifies an indentation value for each parsed result.

```

type tree = Tree {term : pterm; indentT : int; children : tree list}
type offsideTree i = Tree {term : pterm; indentT : { v : int | v > i }; children : (offsideTree i) list}

```

This type defines a tree with three fields:

- A term of type `pterm`.
- The indentation (`indentT`) of a returned parse tree, the refinement constraints on `indentT` requires its value to be greater than `i`.
- A list of sub-parse trees (`children`) for each of the terminals and non-terminals in the current grammar rule’s right-hand side, each of which must also satisfy this refinement.

`Morpheus` additionally automatically generates *qualifiers* like, `indentT`, `children`, etc, for each of the datatype’s constructors and fields with the same name that can be used in type refinements. However, this type is not sufficiently expressive to specify the required safety property for `doBlock` that requires “*the indentation of the parse tree returned by doBlock must be greater than the current value of ist*” because `ist` is an effectful heap variable.

<pre> 1 type α pdo = DoExp of α 2 DoExt of α ... 3 type pterm = 4 PDoBlock of ((pterm pdo) list) 5 let ist = ref 0 ... 6 </pre>	<pre> 21 </pre>
<pre> doBlock : PE^{stexc} {\forall h, I. sel(h, ist) = I} ν : (offsideTree I) result {\forall h, ν, h', I, I'. (ν = Inl(_) => (sel(h, ist) = I \wedge sel(h', ist) = I') => I' = I) \wedge ν = Inr(Err) => (sel(h', inp) \subseteq sel(h, inp)) } </pre>	<pre> indentedDo : PE^{stexc} {\forall h, I. sel(h, ist) = I } ν : tree result {\forall h, ν, h', I, I'. \forall i :int. (i <= I => sel(h', inp) \subseteq sel(h, inp)) \wedge (i > I => indentT(ν) = pos(sel(h, inp)) \wedge children(ν) = nil)} </pre>
<pre> 7 let doBlock = 8 do_m 9 dot \leftarrow reserved "do" 10 ds \leftarrow indentedDoBlock 11 return Tree {term = PDoBlock ds; 12 indentT = indentT(dot); 13 children = (dot :: ds) } 14 </pre>	<pre> 22 23 24 allowed \leftarrow !list 25 i \leftarrow indent 26 if (i <= allowed) then 27 fail("end of block") 28 else 29 do_ 30 </pre>
<pre> do_ : PE^{stexc} {\forall h, I. sel(h, inp) = I} ν : tree result {\forall h, ν, h', I, I'. (ν = Inl(_) => indentT(ν) = pos(sel(h, inp)) children(ν) = nil) \wedge ν = Inr(Err) => (sel(h', inp) \subseteq sel(h, inp)) } </pre>	<pre> 31 32 </pre>
<pre> 15 let do_ = ... 16 </pre>	<pre> 31 32 </pre>
<pre> lookAheadMatches : PE^{pure} {true} ν : bool { [h'=h] } </pre>	<pre> 31 32 </pre>
<pre> 17 lookAheadMatches p = 18 do_m 19 match \leftarrow lookAhead (optional p) 20 return (isJust match) </pre>	<pre> 33 34 35 if (lookAheadMatches (operator)) then 36 do_m 37 operator 38 return (sourceColumn !inp) 39 else 40 return (sourceColumn !inp) </pre>

■ **Figure 11** Morpheus implementation and specifications for a portion of an Idris Do-block with indentation checks, `dom` is a syntactic sugar for Morpheus’s monadic bind. Specifications given in Blue are provided by the parser writer; Gray specifications are inferred by Morpheus. Line number 21 represents the complete multiline type specification.

We can specify a safety property for a `doBlock` parser as shown on line 6 in Figure 11. Again, the type specification in blue are provided by the programmer. The type should be understood as follows: The effect label (`stexc`) defines that the possible effects produced by the parser include `state` and `exc`. The precondition binds the value of the mutable state variable `ist`, a reference to the current indentation level, to `l` via the use of the built-in qualifier `sel` that defines a select operation on the heap [27]. The return type (`offsideTree l result`) obligates the computation to return a parse tree (or a failure) whose indentation must be greater than `l`. The postcondition constraints that the final value of the indentation is to be reset to its value prior to the parse (a *reset* property) when the parser succeeds (case $\nu = \text{Inl}(_)$) or that the input stream `inp` is monotonically consumed when the parser fails (case $\nu = \text{Inr}(\text{Err})$). The types for other parsers in the figure can be specified as shown at lines 14, 21, 32, etc.; these types shown in gray are automatically inferred by Morpheus’s type inference algorithm.

5.3.1 Revisiting the Bug in the Example

The **bug** described in the previous paragraph is unearthed while typechecking the `indentedDo` implementation or the `indentedDoBlock` implementation. We discuss the case for `indentedDo` case here. To verify that `doBlock` satisfies its specification, Morpheus needs to prove that the type inferred for the body of `indentedDo` (lines 22- 29):

1. has a return type that is of the form, `offsideTree I`. Concretely, the indentation of the returned tree must be greater than the initial value of `ist` (i.e. `indentT (ν) > I`).
2. asserts that the final value of `ist` is equal to the initial value.

Goal (1) is required because `indentedDo` is used by `indentedDoBlock` (see Figure 9), which is then invoked by `doBlock`, where its result constructs the value for `children`, whose type is `offsideTree I list`. Goal (2) is required because `doBlock`'s specified post-condition demands it. Type-checking the body for `indentedDo` yields the type shown at line 21. The two conjuncts in the post-condition correspond to the *then* (failure case) and *else* (success case) branch in the parser's body.

The failure conjunct asserts that the input stream is consumed monotonically if the indentation level is greater than `ist`. The success conjunct is the post-condition of the `do_` parser. This inferred type is, however, too weak to prove goal (1) given above, which requires the combinator to return a parse tree that respects the offside rule. The problem is that `indent`'s type (line 32), inferred as:

```
indent : PEstate{true}  $\nu$  : int { $\forall h, \nu, h'. \text{sel}(h', \text{inp}) \subseteq \text{sel}(h, \text{inp})$ }
```

does not allow us to conclude that `indentedDo` satisfies the indentation condition demanded by `doBlock`, i.e., that it returns a well-typed (`offsideTree I`). This is because the type imposes no constraint between the integer `indent` returns and the function's input heap, and thus offers no guarantees that its result gives the position of the first lexeme of the input list.

We can revise `indent`'s implementation such that it does not skip any reserved operators and always returns the position of the first element of the input list, allowing us to track the indentation of every lexeme:

```
indent : PEstate{true}  $\nu$  : int{ $\forall h, \nu, h'. \nu = \text{pos}(\text{sel}(h, \text{inp})) \wedge \text{sel}(h', \text{inp}) \subseteq \text{sel}(h, \text{inp})$ }
let indent = dom s ← !inp
return (sourceColumn s)
```

This type defines a stronger constraint, sufficient to type-check the revised implementation and raise a type error for the original. For this example, Morpheus generated 33 Verification Conditions (VCs) for the revised successful case and 6 VCs for the failing case. We were able to discharge these VCs to the SMT Solver Z3 [7], yielding a total overall verification time of 10.46 seconds in the successful case, and 2.06 seconds in the case when type-checking failed.

This example highlights several key properties of Morpheus verification: The specification language and the type system allows verifying interesting properties over inductive data types (e.g., the `offsideTree` property over the parse trees). It also allows verifying properties dependent on state and other effects such as the *input consumption* property over input streams (`inp`). Secondly, the annotation burden on the programmer is proportional to the complexity of the top-level safety property that needs to be checked. Finally, the similarities between the Haskell implementation and the Morpheus implementation minimize the idiomatic burden placed on Morpheus users.

6 Related Work

Parser Verification. Traditional approaches to parser verification involve mechanization in theorem provers like Coq or Agda [29, 6, 10, 21, 39, 23, 15]. These approaches trade-off both automation and expressiveness of the grammar they verify to prove full correctness. Consequently, these approaches cannot verify safety properties of data-dependent parsers, the subject of study in this paper. For instance, RockSalt [29] focuses on regular grammars, while [21, 10] present interpreters for parsing expression grammars (without nondeterminism) and limited semantic actions without data dependence. Jourdan et al. [16] gives a certifying compiler for LR(1) grammars, which translates the grammar into a pushdown automaton and a certificate of language equivalence between the grammar and the automaton. More recently CoStar [23] presents a fully verified parser for the ALL(*) fragment mitigating some of the limitations of the above approaches. However, unlike *Morpheus*, CoStar does not handle data-dependent grammars or user-defined semantic actions.

Deductive synthesis techniques for parsers like Narcissus [8] and [37] focus mainly on tag-length-payload, binary data formats. Narcissus [8] provides a Coq framework (an `encode_decode` tactic) that can automatically generate correct-by-construction encoders and decoders from a given user format input, albeit for a restricted class of parsers. Notably, the system is not easily extensible to complex user-defined data-dependent formats such as the examples we discuss in *Morpheus*. This can be attributed to the fact that the underlying `encode_decode` Coq tactic is complex and brittle and may require manual proofs to verify a new format. In contrast, *Morpheus* enables useful verification capabilities for a larger class of parsers, albeit at the expense of automatic code generation and full correctness. Writing a safe parser implementation for a user-defined format in *Morpheus* is no more difficult than manually building the parser in any combinator framework with the user only having to provide an additional safety specification. EverParse [37] likewise focuses mainly on binary data formats, guaranteeing full-parser correctness, albeit with some expressivity limitations. For example, it does not support user-defined semantic actions or global data-dependences for general data formats. Compared to these efforts, the properties *Morpheus* can validate are more high-level and general. E.g., “non-overlapping of two lists of strings” in a C-decl parser; “layout-sensitivity properties”, etc.. Verifying these properties requires reasoning over a challenging combination of rich algebraic data types, mutable states, and higher-order functions.

[22] also explore types for parsing, defining a core type-system for context-free expressions. However, their goals are orthogonal to *Morpheus* and are targeted towards identifying expressions that can be parsed unambiguously.

Data-dependent and Stateful Parsers. *Morpheus* allows writing parsers for data-dependent and stateful parsers. There is a long line of work aimed at writing such parsers [14, 1, 2, 24]. None of these efforts, however, provide a mechanism to reason about the parsers they can express. Further, many of these systems are specialized for a particular class/domain of problems, such as [14] for data-dependent grammars with trivial semantic actions, or [1] for indentation sensitive grammars, etc. *Morpheus* is sufficiently expressive to both write parsers and grammars discussed in many of these approaches, as well as verifying interesting safety properties. Indeed, several of our benchmarks are selected from these works. In contrast, systems such as [14] argue about the correctness of the input parsed against the underlying CFG, a property challenging to define and verify as a *Morpheus* safety property, beyond simple string-patterns and regular expressions. We leave the expression of such grammar-related properties in *Morpheus* as a subject for future work.

Refinement Types. Our specification language and type system builds over a refinement type system developed for functional languages like Liquid Types [38] or Liquid Haskell [43]. Extending Liquid Types with *bounds* [42] provides some of the capabilities required to realize data-dependent parsing actions, but it is non-trivial to generalize such an abstraction to complex parser combinators found in Morpheus with multiple effects and local reasoning over states and effects.

Effectful Verification. Our work is also closely related to dependent-type-based verification approaches for effectful programs based on monads indexed with either pre- and post-conditions [31, 32] or more recently, predicate monads capturing the weakest pre-condition semantics for effectful computations [41]. As we have illustrated earlier, the use of expressive and general dependent types, while enabling the ability to write rich specifications (certainly richer than what can be expressed in Morpheus), complicates the ability to realize a fully automated verification pathway.

Verification using natural proofs [36] is based on a mechanism in which a fixed set of proof tactics are used to reason about a set of safety properties; automation is achieved via a search procedure over in this set. This idea is orthogonal to our approach where we rather utilize the restricted domain of parsers to remain in a decidable realm. Both our effort and these are obviously incomplete. Another line of work verifying effectful specifications use characteristic formulae [5]; although more expressive than Morpheus types, these techniques do not lend themselves to automation.

Local Reasoning over Heaps. Our approach to controlling aliasing is distinguished from substructural typing techniques such as the ownership type system found in Rust [17]. Such type systems provide a much richer and more expressive framework to reason about memory and effects, and can provide useful guarantees like memory safety and data-race freedom etc. Since our DSL is targeted at parser combinator programs which generally operate over a much simplified memory abstraction, we found it unnecessary to incorporate the additional complexity such systems introduce. The integration of these richer systems within a refinement type framework system of the kind provided in Morpheus is a subject we leave for future work.

Parser Combinators. There is a long line of work implementing Parser Combinator Libraries and DSLs in different languages [11]. These also include those which provide a principled way for writing stateful parsers using these libraries [1, 24]. As we have discussed, none of these libraries provide an automated verification machinery to reason about safety properties of the parsers. However, since they allow the full expressive power of the host language, they may, in some instances, be more expressive than Morpheus. For example, Morpheus does not allow arbitrary user-defined higher-order functions and builds only on the core API discussed earlier. This may require a more intricate definition for some parsers compared to traditional libraries. For example, traditional parser combinator libraries typically define a higher-order combinator like `many_fold_apply` with the following signature and use this combinator to concisely define a *Kleene-star* parser:

```
many_fold_apply : f : ('b → 'a → 'b) → (a : 'a) → (g : 'a → 'a) → p : ('a, 's) t → ('b, 's) t
let many p = many_fold_apply (fun xs x → x :: xs) [] List.rev p
```

Contrary to this, in Morpheus, we need to define Kleene-star using a more complex, lower-level fixpoint combinator.

7 Conclusions

This paper presents Morpheus, a deeply-embedded DSL in OCaml that offers a restricted language of composable effectful computations tailored for parsing and semantic actions and a rich specification language used to define safety properties over the constituent parsers comprising a program. Morpheus is equipped with a rich refinement type-based automated verification pathway. We demonstrate Morpheus’s utility by using it to implement a number of challenging parsing applications, validating its ability to verify non-trivial correctness properties in these benchmarks.

References

- 1 Michael D. Adams and Ömer S. Ağacan. Indentation-sensitive parsing for parsec. In *SIGPLAN Notices*, volume 49(12), pages 121–132, New York, NY, USA, September 2014. Association for Computing Machinery. doi:10.1145/2775050.2633369.
- 2 Ali Afrozeh and Anastasia Izmaylova. One parser to rule them all. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 151–170, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2814228.2814242.
- 3 Angstrom. Angstrom parser-combinator library, 2021. URL: <https://github.com/inhabitedtype/angstrom>.
- 4 Edwin Brady. Idris: Implementing a dependently typed programming language. In *Proceedings of the 2014 International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP ’14, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2631172.2631174.
- 5 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. *SIGPLAN Not.*, 46(9):418–430, September 2011. doi:10.1145/2034574.2034828.
- 6 Nils Anders Danielsson. Total parser combinators. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP ’10*, pages 285–296, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1863543.1863585.
- 7 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 8 Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341686.
- 9 DNS. Domain names – Implementation and specification, 1987. Network Working Group. URL: <https://www.rfc-editor.org/rfc/rfc1035>.
- 10 J. Gross and Adam Chlipala. Parsing parsers a pearl of (dependently typed) programming and proof, 2015.
- 11 HaskellWiki. Parsec – HaskellWiki, 2021. [Online; accessed 7-July-2022]. URL: <https://wiki.haskell.org/index.php?title=Parsec&oldid=64649>.
- 12 Graham Hutton and Erik Meijer. Monadic parser combinators, September 1999.
- 13 *Documentation for the Idris Language*, 2017. URL: <https://docs.idris-lang.org/en/latest/index.html>.
- 14 Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’10, pages 417–430, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1706299.1706347.
- 15 Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating lr(1) parsers. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 397–416, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

- 16 Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating lr(1) parsers. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 397–416, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 17 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158154.
- 18 Gowtham Kaki and Suresh Jagannathan. A relational framework for higher-order shape analysis. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 311–324, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2628136.2628159.
- 19 Mark Karpov. Megaparsec: Monadic Parser Combinators, 2022. URL: <https://github.com/mrkkp/megaparsec>.
- 20 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 633–645, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2535838.2535846.
- 21 Adam Koprowski and Henri Binszok. Trx: A formally verified parser editor=Gordon, Andrew D., interpreter. In *Programming Languages and Systems*, pages 345–365, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 22 Neelakantan Krishnaswami and Jeremy Yallop. A typed, algebraic approach to parsing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 379–393, June 2019. doi:10.1145/3314221.3314625.
- 23 Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. Costar: A verified all(*) parser. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 420–434, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454053.
- 24 Nicolas Laurent and Kim Mens. Taming context-sensitive languages with principled stateful parsing. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 15–27, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2997364.2997370.
- 25 Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, University of Utrecht, July 2001. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007. URL: <https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/>.
- 26 Simon Marlow. Haskell 2010 language report, 2010. URL: <https://www.haskell.org/onlinereport/haskell2010/>.
- 27 J. McCarthy. *Towards a Mathematical Science of Computation*, pages 35–56. Springer Netherlands, Dordrecht, 1993. doi:10.1007/978-94-011-1793-7_2.
- 28 Ashish Mishra and Suresh Jagannathan. Morpheus: Automated safety verification of data-dependent parser combinator programs, 2023. doi:10.48550/arXiv.2305.07901.
- 29 Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: Better, faster, stronger sfi for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254111.
- 30 Max Murato. MParser, A Simple Monadic Parser Combinator Library, 2021. URL: <https://github.com/murmour/mparser>.
- 31 Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. *SIGPLAN Not.*, 41(9):62–73, September 2006. doi:10.1145/1160074.1159812.

- 32 Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. *SIGPLAN Not.*, 43(9):229–240, September 2008. doi:10.1145/1411203.1411237.
- 33 Meredith L. Patterson. Hammer primer, 2015. URL: <https://github.com/sergeybratus/HammerPrimer>.
- 34 PDF. Iso 32000 (pdf), 2008. PDF Association. URL: <https://www.pdfa.org/resource/iso-32000-pdf/pdf-2>.
- 35 PKWare. zip file format specification, 2020. URL: <https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>.
- 36 Xiaokang Qiu, Pranav Garg, Andrei Ştefănescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. *SIGPLAN Not.*, 48(6):231–242, June 2013. doi:10.1145/2499370.2462169.
- 37 Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19*, pages 1465–1482, USA, 2019. USENIX Association.
- 38 Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08*, pages 159–169, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1375581.1375602.
- 39 Kathleen Fisher Sam Lasser, Chris Casinghino and Cody Roux. A verified ll(1) parser generator. In *ITP*, 2019.
- 40 Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009*. IEEE Computer Society, January 2008.
- 41 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 387–398, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491956.2491978.
- 42 Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 48–61, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2784731.2784745.
- 43 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282. ACM, 2014. doi:10.1145/2628136.2628161.
- 44 Philip Wadler. Monads for functional programming. In Manfred Broy, editor, *Program Design Calculi*, pages 233–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
- 45 Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, January 2003. doi:10.1145/601775.601776.

Automata Learning with an Incomplete Teacher

Mark Moeller  

Cornell University, Ithaca, NY, USA

Thomas Wiener 

Cornell University, Ithaca, NY, USA

Alaia Solko-Breslin  

University of Pennsylvania, Philadelphia, PA, USA

Caleb Koch 

Stanford University, CA, USA

Nate Foster  

Cornell University, Ithaca, NY, USA

Alexandra Silva  

Cornell University, Ithaca, NY, USA

Abstract

The preceding decade has seen significant interest in use of active learning to build models of programs and protocols. But existing algorithms assume the existence of an idealized oracle – a so-called Minimally Adequate Teacher (MAT) – that cannot be fully realized in practice and so is usually approximated with testing. This work proposes a new framework for active learning based on an incomplete teacher. This new formulation, called iMAT, neatly handles scenarios in which the teacher has access to only a finite number of tests or otherwise has gaps in its knowledge. We adapt Angluin’s L^* algorithm for learning finite automata to incomplete teachers and we build a prototype implementation in OCaml that uses an SMT solver to help fill in information not supplied by the teacher. We demonstrate the behavior of our iMAT prototype on a variety of learning problems from a standard benchmark suite.

2012 ACM Subject Classification Theory of computation → Active learning

Keywords and phrases Finite Automata, Active Learning, SMT Solvers

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.21

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.21>

Software (Source Code): <https://github.com/cornell-pl/nerode-public>

archived at [swh:1:dir:c5e04db5d43057cfb4a080a987539d73486e676a](https://swh.1:dir:c5e04db5d43057cfb4a080a987539d73486e676a)

Funding This material is based upon work supported by the Office of Naval Research under Contract No. N68335-22-C-0411.

Acknowledgements We thank Marijn Heule, Martin Leucker, and Arlindo Oliveira for their efforts in providing us access to their code and benchmarks. We also thank Akshat Singh and Sheetal Athrey, with whom this project began as an undergraduate research project.

1 Introduction

Automata are among the most basic structures in computer science, yet they continue to offer profound insights for modeling and analyzing systems. Recent years have seen renewed interest in the problem of *closed-box inference* of automata, often motivated by applications in verification and security – see [33] for an overview.



© Mark Moeller, Thomas Wiener, Alaia Solko-Breslin, Caleb Koch, Nate Foster, and Alexandra Silva;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 21; pp. 21:1–21:30



Leibniz International Proceedings in Informatics
LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Many of the algorithms developed for closed-box inference are based on the *minimally adequate teacher* (MAT) framework, in which a Learner interacts with a Teacher (also sometimes referred to as an oracle) in an attempt to learn a finite automaton known by the Teacher. The Learner can pose two types of queries to the Teacher: membership queries (“does the automaton accept a given input word?”) and equivalence queries (“is a given automaton correct?”). It turns out that these primitives are sufficient to show that the MAT framework terminates and produces the correct automaton. Numerous variants of the basic algorithm have been proposed (e.g., L^* [2], KV [19], TTT [18] or $L^\#$ [34]), using different data structures to collect and organize gathered information. The MAT framework has also been instantiated for other kinds of automata – e.g. Mealy machines, weighted automata [5], and nominal automata [28] to name a few.

Contrary to its name, however, in many practical settings, even finding a Minimally Adequate Teacher is challenging. For example, consider learning a model of a closed-box system. How would the Teacher determine whether a given automaton supplied by the Learner captures the behavior of the closed-box system? The best the Teacher can do is check whether they agree on a finite set of tests. Likewise, in settings where the Teacher only has access to a set of positive and negative examples (also known as passive learning) it is unclear how to answer membership queries for inputs that lie outside of the example set.

This paper presents an alternative to MAT: the *incomplete* Minimally Adequate Teacher (iMAT) framework. An iMAT is allowed to answer “don’t care” in response to membership queries and it does not answer equivalence queries at all. We show that Angluin’s classic L^* algorithm can be instantiated with an incomplete Teacher, by using an SMT solver to help fill in the missing information without needlessly expanding the size of the automaton. More precisely, we show – under modest assumptions – that our algorithm infers the minimal automaton compatible with the information provided by the Teacher. We present an OCaml prototype and demonstrate its behavior on well-known benchmarks [23, 29].

We see the framework developed in this paper as a first step towards building connections between several different areas. As we explain later, iMAT provides a bridge between active and passive learning. In active learning, additional information can always be gathered (i.e., from the Teacher), whereas in passive learning, the assumption is that all of the information that will be used for learning has been gathered in advance. With iMAT, one can use the passive information as an incomplete Teacher and proceed to learn the automaton using active techniques like L^* . A lack of perfect information about the language in question also arises in program synthesis, in particular in the “programming by example” paradigm [7, 22, 23, 25, 36]. Here, the goal is to infer a model that is correct for the examples, and hope that it generalizes to other scenarios – i.e., that the examples are somehow representative of a more general pattern. We explore this connection in our use of benchmarks from ALPHAREGEX [23].

The problem of using finite information to infer a DFA was first studied by Gold in the 1970s [14], and has since been studied by many others [32, 30, 21, 20]. Gold showed that finding a DFA with the minimum number of states is NP-complete [15]. As iMAT subsumes DFA inference from finite data, its scalability is necessarily limited.

Others have studied problems similar to iMAT in prior work, either in their use of incomplete information or SAT solvers [8, 29, 17, 16, 24]. Leucker and Neider’s paper [24] includes an “inexperienced teacher” and surveys several learners with similar capabilities as ours. Section 10 presents a detailed survey of their paper and other related work. No previous work has studied iMAT in full generality, including: formulating the problem, establishing correctness, building an open-source implementation, and exploring alternate formulations. Hence, compared to prior work, this paper makes the following contributions:

1. We review the MAT framework (Section 2) which sets the stage for our formulation of active learning based on an incomplete Teacher (Section 3). We also adapt the notion of a Learner, giving it access to a Solver.
 2. We instantiate the iMAT framework in the context of DFAs, using an SMT solver to fill in missing information (Section 4).
 3. We prove that the Learner terminates and returns a minimal automaton compatible with the known information (Section 5).
 4. We show how to modify our learning algorithm in the presence of the more limited Teacher that uses a simpler interface for compatibility checks (Section 6), and prove a (partial) correctness result (Section 7).
 5. We implement the Learner in OCaml and present optimizations to accelerate the search for a DFA. We evaluate our proposed optimizations with an ablation study, and present performance data on a standard benchmark suite [23, 29].
- Next, we review MAT and L^* , to set the stage for the iMAT framework in later sections.

2 The MAT framework

In the MAT framework, a Learner seeks to discover an unknown language by interacting with a Teacher who can answer two types of queries:

Membership: The Learner sends a word u to the Teacher, who answers “yes” if u belongs to the language or “no” if it does not.

Equivalence: The Learner sends a hypothesis automaton \mathcal{H} to the Teacher, who either confirms that \mathcal{H} is correct or provides a counterexample.

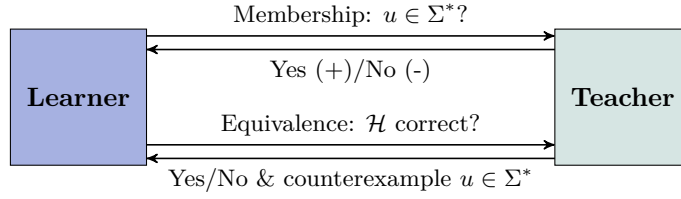
The MAT framework has been the basis for active model learning since its introduction in the late 1980s, providing the basis for a variety of algorithms, data structures, and proof techniques (e.g. L^* [2], KV [19], TTT [18], L^\sharp [34], etc.). The rest of this section presents Angluin’s L^* algorithm, which introduces the notation and concepts used in this paper.

2.1 Strings, Languages, and Automata

Fix an alphabet of symbols, Σ . We let $a \in \Sigma$ denote an arbitrary symbol and ε the empty string. For strings (also called words) $u, v \in \Sigma^*$, we write uv for the concatenation of u and v (we will occasionally write $u.v$ for emphasis). We use capital letters $U, V \subseteq \Sigma^*$ to denote languages – i.e. sets of strings. Concatenation of languages U, V is written $U \cdot V = \{uv \mid u \in U, v \in V\}$. The set of prefixes for a string s is written $\text{prefixes}(s) = \{u \mid s = uv, u, v \in \Sigma^*\}$. Likewise, $\text{suffixes}(s) = \{v \mid s = uv, u, v \in \Sigma^*\}$. We sometimes even take prefixes of a set $S \subseteq \Sigma^*$, written $\text{prefixes}(S) = \{u \mid u \in \text{prefixes}(s), s \in S\}$. A set $S \subseteq \Sigma^*$ is called *prefix-closed* (*suffix-closed*) if $S = \text{prefixes}(S)$ ($S = \text{suffixes}(S)$). Finally, we denote the symmetric difference of two sets by $S \oplus S' = S - S' \cup S' - S$.

An important class of languages is that of *regular languages*, which are the languages accepted by deterministic finite automata (DFAs). A DFA is a five-tuple $D = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is the alphabet, $\delta: Q \times \Sigma \rightarrow Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ are the accepting states. The transition function is extended to strings inductively by $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$, where for any $q \in Q$, $\hat{\delta}(q, \varepsilon) = q$ and for any $a \in \Sigma, u \in \Sigma^*$, $\hat{\delta}(q, au) = \hat{\delta}(\delta(q, a), u)$. A string u is accepted by the automaton D iff $\hat{\delta}(q_0, u) \in F$. Moreover, the language accepted by D , written $L(D)$, is the set of all such strings: $L(D) = \{u \in \Sigma^* \mid \hat{\delta}(q_0, u) \in F\}$.

21:4 Automata Learning with an Incomplete Teacher



■ **Figure 1** The Minimally Adequate Teacher framework.

Regular languages have unique minimal representatives: for every regular language, there is a unique DFA with a minimal number of states. There are different ways to build such minimal DFA corresponding to a regular language, but the one that is used to prove correctness of L^* is based on so-called Myhill-Nerode equivalence classes. Given a language L and a word $s \in \Sigma^*$, the Myhill-Nerode equivalence class of s , denoted $[s]_L$ is the set of all words s' satisfying: $s \equiv_L s' \stackrel{\Delta}{\iff} \forall e \in \Sigma^* \cdot (se \in L \iff s'e \in L)$. Myhill-Nerode equivalence classes provide an alternative characterization of regular languages: a language is regular iff it has a finite number of Myhill-Nerode equivalence classes. Furthermore, there is a one-to-one correspondence between the states of the minimal automaton accepting a regular language L and its Myhill-Nerode equivalence classes.

2.2 L^* : Data Structures

L^* is an algorithm that implements a Learner according to the interfaces in the MAT framework (see Figure 1). The core data structure used in the algorithm is an observation table, which records the information gathered from membership queries and the counterexamples obtained from equivalence queries.

► **Definition 1** (Observation Table). *Given a language $L \subseteq \Sigma^*$, an observation table (wrt L) is a triple (S, E, T) , where:*

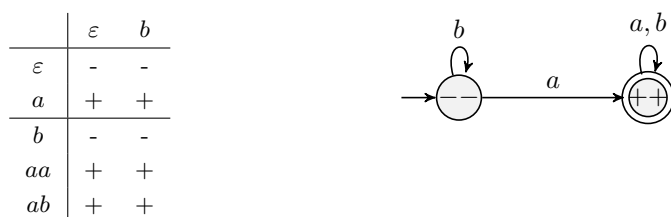
- $S \subseteq \Sigma^*$ is a prefix-closed set of “accessor strings”;
- $E \subseteq \Sigma^*$ is a suffix-closed set of “distinguishing strings”;
- $T: (S \cup S \cdot \Sigma) \cdot E \rightarrow \{+, -\}$ is a map on a finite set of words defined by

$$T(u) = \begin{cases} + & u \in L \\ - & u \notin L \end{cases}$$

Note that once L is fixed, T is fully determined by S and E , so sometimes we refer to the observation table as simply (S, E) . We can think of $L \subseteq \Sigma^*$ as a function $L: \Sigma^* \rightarrow \{+, -\}$ and then simply write $T(u) = L(u)$. Every u in the domain of T is obtained as a concatenation of a string $s \in S \cup S \cdot \Sigma$ and $e \in E$.

Intuitively, one can think of an observation table as a snapshot of a language L . When L is regular, we will show how the table is organized in a way that its rows can be used to recover the Myhill-Nerode equivalence classes of L , and therefore the minimal deterministic automaton. The use of accessor strings and distinguishing strings to name the elements of S and E will become clear when we explain how to build a DFA from a table.

Consider the regular language $L = \{w \in \{a, b\}^* \mid w \text{ has at least one } a\}$, and sets $S = \{\varepsilon, a\}$, and $E = \{\varepsilon, b\}$. The observation table (S, E) is given on the left in Figure 2. When depicting the table, we divide it into an upper part and a lower part. The rows in the upper part are labelled by strings in S , whereas the rows of the lower part are labelled by strings in $S \cdot \Sigma - S = \{sa \mid s \in S, a \in \Sigma, sa \notin S\}$. The strings in E label the columns of the table. The entries of the table correspond to the function T .



■ **Figure 2** An observation table and corresponding automaton.

It is important to note that depicting the finite map $T: (S \cup S \cdot \Sigma) \cdot E \rightarrow \{+, -\}$ as a table leads to some repetition. For example, the entry in row a , column b must match the entry in row ab , column ε – i.e., since $a.b = ab.\varepsilon$ we also have $T(a.b) = T(ab.\varepsilon)$. It will be convenient to access the rows of the table as follows:

$$\text{row}: S \cup S \cdot \Sigma \rightarrow E \rightarrow \{+, -\} \quad \text{row}(s)(e) = T(se).$$

We will sometimes abuse notation and apply row to a set of strings to obtain a set of rows. For example, in the table above $\text{row}(S)$ is the set of distinct rows in the upper part of the table – $\text{row}(S) = \{\{\varepsilon \mapsto +, b \mapsto +\}, \{\varepsilon \mapsto -, b \mapsto -\}\}$. Additionally, when E is clear from the context, we will sometimes omit the column indices, writing $\text{row}(S) = \{++, --\}$.

To build a DFA from an observation table, we will use as states the upper rows of the table, but the well-definedness of the construction requires two properties to be satisfied¹.

► **Definition 2** (Closedness and Distinctness). *A table (S, E, T) is closed if for each string $s \in S$ and letter $a \in \Sigma$, we have $\text{row}(sa) \in \text{row}(S)$. It is distinct if all the rows labelled by $s \in S$ are distinct: $s, s' \in S \Rightarrow \text{row}(s) \neq \text{row}(s')$.*

Note that the example table above is closed and distinct. We can now make the connection between observation tables and finite automata precise.

► **Definition 3** (DFA associated with a table). *Given a closed and distinct table (S, E, T) , with distinct rows in S , we associate a DFA, $\mathcal{D}(S, E, T) = (Q, \Sigma, \delta, q_0, F)$, where:*

$$Q = \text{row}(S) \quad \delta(\text{row}(s), a) = \text{row}(sa) \quad q_0 = \text{row}(\varepsilon) \quad F = \{\text{row}(s) \mid \text{row}(s)(\varepsilon) = +\}.$$

The above definition relies on the fact that S is prefix-closed and E is suffix-closed and so both contain ε . Moreover, the transition function δ is well-defined whenever the table is closed and distinct. The first property ensures that $\text{row}(sa) \in Q$, whereas the second ensures the well definedness of $\delta(\text{row}(s), a)$.

The DFA corresponding to the example observation table is depicted on the right in Figure 2. The start state is indicated with an incoming arrow, and final state with a double circle. Each state is labeled by its unique row vector.

2.3 L^* learner

We are now ready to present L^* , the algorithm in which a Learner incrementally builds an observation table based on interactions with a Teacher as depicted in Figure 1. The L^* Learner is shown in Figure 3. It starts with $S = E = \{\varepsilon\}$ and then explores potential new

¹ Readers familiar with L^* will notice we jettisoned consistency for a simpler property – keeping the upper rows of the table distinct. This optimization is due to Maler and Pnueli [27].

1. Initialize $S = \{\varepsilon\}$, and $E = \{\varepsilon\}$
2. While (S, E, T) is not closed, do:
 - $S \leftarrow S \cup \{sa\}$,
 - where $s \in S, a \in \Sigma$, but $\text{row}(sa) \notin \text{row}(S)$.
3. Conjecture $M = \mathcal{D}(S, E, T)$:
 - a. If M is correct, halt.
 - b. Otherwise, $E \leftarrow E \cup \text{suffixes}(c)$, where c is the provided counterexample. Goto 2.

■ **Figure 3** Angluin’s L^* (Maler-Pnueli version). Whenever S and E change, T is updated using membership queries. Note the extensions of S and E preserve distinctness as an invariant.

rows by examining the rows in the lower part of the table (labelled by strings in $S \cdot \Sigma - S$). If no new rows are found (i.e., as compared to the ones in $\text{row}(S)$), then we conclude that the table is closed. If, on the other hand, a row in the lower part of the table does not appear in the upper part, there is a closedness defect, which can be repaired by adding another string to S . In essence, this repair moves a row from the lower part of the table to the upper part. Accordingly, we generate new rows in the lower part until there is a row in the table for each $s \in S\Sigma$. Once the Learner finds a closed table (distinctness is maintained by construction), it poses an equivalence query to the Teacher. If the hypothesis is wrong the Teacher returns a counterexample which is used to extend the columns of the table.

L^* can also be understood as incrementally discovering the Myhill-Nerode equivalence classes for the Teacher’s language \mathcal{L} , which correspond to the states of the minimal DFA. Since the start state always corresponds to the equivalence class for the empty string ε , it starts with the observation table where $S = \{\varepsilon\}$ and $E = \{\varepsilon\}$. Each hypothesis is a refinement of the previous guess, getting closer and closer to the correct minimal automaton. Termination of the algorithm follows as every closedness defect repaired or counterexample processed, results in an automaton with more states than the previous hypothesis.

2.4 L^* Example

Suppose we choose $\Sigma = \{a\}$ and the Teacher knows the language L described by the regular expression $a(aaa)^*$. The Learner begins by building the observation table below, on the left:

	ε
ε	-
a	+

	ε
ε	-
a	+
aa	-

(1)

The table on the left is not closed, because the row for a is not represented in the upper part of the table, hence the Learner extends S and this yields the table above in the middle. This table is closed so the Learner conjectures the corresponding DFA on the right.

The Teacher returns counterexample aaa (which is accepted by the automaton in Equation (1) but should be rejected).² The Learner processes this counterexample by extending E with its suffixes $\{\varepsilon, a, aa, aaa\}$, yielding the table on the left below:

² In general, there is no requirement for the Teacher to return the shortest counterexample and, indeed, the original complexity analysis of L^* takes into account that longer counterexamples might be returned resulting in larger than needed tables. Note that, however, minimality is never compromised as equal rows will be mapped to the same state in the automaton.

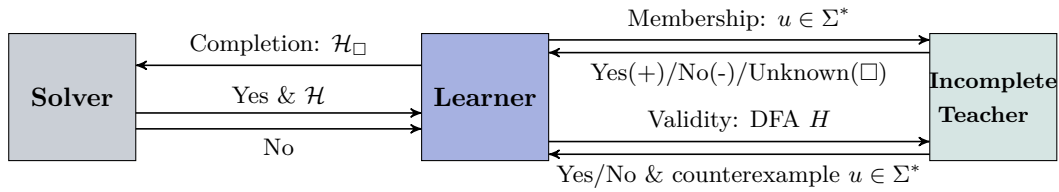


Figure 4 The Incomplete Minimally Adequate Teacher framework (iMAT).

	ε	a	aa	aaa
ε	-	+	-	-
a	+	-	-	+
aa	-	-	+	-

	ε	a	aa	aaa
ε	-	+	-	-
a	+	-	-	+
aa	-	-	+	-
aaa	-	+	-	-

The table on the left above is not closed, as row(aa) is different from all other rows. To fix this, the Learner adds aa to S yielding the table above, on the right. This table is closed and distinct, so the Learner makes the following conjecture which the Teacher accepts:



Angluin [2] showed that the L^* learner finds the minimal DFA for the target language using a polynomial number of membership queries – results which also apply to the Maler-Pnueli version. However, this property relies on the assumption that the Teacher can answer membership and equivalence queries.

The rest of this paper is devoted to introducing an alternative to the MAT framework, in which the Teacher is incomplete – it might not have answers to all the membership questions – but there is a third agent in the process that can help the Learner guess the missing information. We will then devise, implement, and evaluate an algorithm based on L^* .

The challenges in devising and implementing such an algorithm are two-fold: on the one hand, since we do not have access to arbitrary membership queries, we will be building a table that has *holes*; on the other hand, the notion of minimality is now with respect to the existing information. Hence, progress towards this automaton is not as simple as in L^* , requiring the use of heuristics to guess the missing information while still minimizing the size of the final automaton.

3 The iMAT Framework

This section introduces iMAT, a new framework for automata learning in which the Teacher is *incomplete* (see Figure 4). In iMAT, the Learner still wants to infer a regular language L , but the teacher only has partial information about the language. In particular, instead of holding an explicit language L , the Teacher is assumed to hold disjoint, possibly infinite sets of positive L_+ and L_- negative examples. The Learner seeks to find any language L such that $L_+ \subseteq L$ and $L_- \cap L = \emptyset$. In the literature, such an L is said to *separate* L_+ and L_- [8]. We assume that at least one regular L exists, but there may be several.³

³ Note that we do *not* require that L_+ or L_- be regular. For example, neither $L_+ = a^n b^n, n > 0$ nor $L_- = b^n a^n, n > 0$ are regular, but they are separated by $L = a^* b^*$.

-
1. $\text{worklist} \leftarrow [(\{\varepsilon\}, \{\varepsilon\})]$
 2. Call to SMT solver to fill in \square s with $+$ or $-$ in $\text{hd}(\text{worklist})$.
 - a. if UNSAT then:
 - i. $\text{worklist} \leftarrow \text{tl}(\text{worklist}) @ [(S \cup \{s'\}, E) \mid s' \in S \cdot \Sigma - S]$
 - ii. Goto step (2).
 - b. if SMT solver returns table (S, E) , build the corresponding DFA and make a validity query.
 - i. If the validity query succeeds, return the DFA.
 - ii. Otherwise, get a counterexample c , set $E' = E \cup \text{suffixes}(c)$, build table with \square 's (S, E') and set $\text{worklist} \leftarrow (S, E') :: \text{worklist}$; goto step (2).
-

■ **Figure 5** L^* with blanks algorithm, L_{\square}^* . We use '@' and '::' to denote standard list operations (i.e., append and cons). As before, we omit membership queries; these occur whenever S and E are changed. Since observation tables are fully determined by S and E , we elide T .

The iMAT framework has three components: a Learner, an incomplete Teacher, and a Solver. The Teacher answers two types of queries, like MAT, but with weaker assumptions: **Membership:** The Learner sends a word u to the incomplete Teacher, who answers with “yes” (+), “no” (-), or “unknown” (\square).

Validity: Given a hypothesis automaton, \mathcal{H} , the teacher determines whether $L_+ \subseteq L(\mathcal{H})$ and $L_- \cap L(\mathcal{H}) = \emptyset$. If so, it returns *None*, otherwise, it returns a counterexample string either in $L_+ - L(\mathcal{H})$ or in $L_- \cap L(\mathcal{H})$.

The other component in the iMAT framework is the Solver which the Learner can ask for help in completing the hypothesis construction. The solver answers just one type of query:

Completion: The Learner sends an incomplete hypothesis to the solver and asks whether there is a way to complete it into a full automaton. The Solver either says “yes” and returns a complete hypothesis or “no”.

iMAT’s Validity query is obviously similar to MAT’s Equivalence query, but it is different in an important way: an incomplete Teacher cannot return any string not in L_+ or L_- as counterexamples as it lacks information about those strings. Hence, a Validity query returns “yes” whenever there is no evidence against the hypothesis, which is subtly different than confirming the hypothesis directly. So while Validity may seem just as complex as Equivalence, we include it here as a first step toward more practical queries.

► **Remark 4.** Note that if the incomplete Teacher happens to be a complete oracle, then the iMAT setup can be used for MAT. Consider an iMAT instance where (i) the membership queries always return “yes” or “no” and (ii) the validity query returns a counterexample iff one exists. It follows that the hypothesis \mathcal{H} can be built without ever calling the Solver (and with the same membership query complexity as MAT) and Validity queries correspond precisely to Equivalence queries.

4 L_{\square}^* : an iterative iMAT Learner

We now present an iMAT Learner, closely following the approach used in L^* . In essence, looking back at the schema in Figure 4 we want to devise a learner that exploits the loop of membership-validity queries to promote a steady construction of a minimum-size automaton with the minimum amount of information. For the Solver component of iMAT we use an SMT solver (i.e., Z3 [11]).

The Learner holds a (sorted, in increasing size) worklist of candidate observation tables with \square 's. The algorithm works by constructing a worklist of potential hypotheses of increasing size, starting from a table with just ε as row and column labels.

We iteratively take each of these candidate tables and use an SMT solver to attempt filling in the \square 's (Completion) so that the table is closed and distinct (see Figure 6). Based on the result of the SMT query, we will either pass the completed table to the Teacher (Validity) or try another candidate table. If the Validity query succeeds, we return it. Otherwise, if we get a counterexample c , we refine the current hypothesis table by adding all the suffixes of c to E , and add the table (with \square 's) back to the worklist. The resulting iterative algorithm, L^* with blanks (L^*_\square), is shown in Figure 5.

Given an observation table, (S, E, T) :

1. Declare a Boolean variable b_s for each string $s \in (S \cup S \cdot \Sigma) \cdot E$ for which $T(s) = \square$.
2. For strings in L_+ or L_- , we fix the Boolean constraints:

$$\bigwedge_{s \in L_+} b_s = \text{true} \quad (\text{Positive Evidence})$$

$$\bigwedge_{s \in L_-} b_s = \text{false} \quad (\text{Negative Evidence})$$

3. Define a predicate to assert rows equal:

$$\text{Eq}(s, s') \triangleq \bigwedge_{e \in E} b_{se} = b_{s'e} \quad (\text{Rows equal})$$

4. Assert the table is closed:

$$\bigwedge_{s' \in S \cdot \Sigma - S} \bigvee_{s \in S} \text{Eq}(s, s') \quad (\text{Closed})$$

5. Define a predicate to represent the property that a row is unique (more precisely, that it is the first time the row appears):

$$\text{Unique}(s) \triangleq \bigwedge_{s' \in S: s' <_{\text{lex}} s} \neg \text{Eq}(s, s') \quad (\text{Row unique})$$

6. Declare a Boolean u_s for each string $s \in S$. We set u_s to true if row(s) is the first time that row appears:

$$u_s = \text{Unique}(s)$$

7. Assert the uniqueness of the rows in S (to maintain distinctness invariant):

$$\bigwedge_{s \in S} \text{Unique}(s)$$

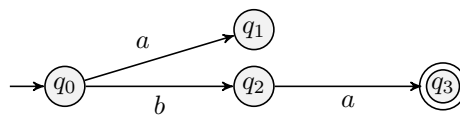
■ **Figure 6** SAT encoding for table constraints.

21:10 Automata Learning with an Incomplete Teacher

When the SMT solver finds that the instance is unsatisfiable, we know that the table cannot be made closed, but we do not know which s from $S \cdot \Sigma - S$ we need to promote to S to fix things as we would in classic L^* . So we make progress by extending separate “copies” of S with each string in $S \cdot \Sigma - S$, and adding these tables to the worklist.

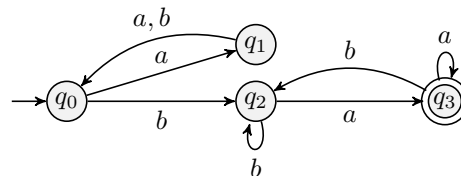
► **Remark 5.** By adopting the Maler-Pnueli variation of L^* , the rows in S will always be distinct and correspond to nodes in a prefix tree. The worklist of tables with \square 's is providing an enumeration of the search space of prefix trees that correspond to DFAs of increasing size. The SMT solver will essentially be checking for every entry of the worklist whether, for a given prefix tree corresponding to a table, the other transitions not in the tree can be filled in to be consistent with the data. For example, consider the following snapshot of a table in the worklist and the corresponding prefix tree:

	ε	a	aa	baa
ε	\square	-	\square	+
a	-	\square	-	\square
b	-	+	+	+
ba	+	+	+	\square
aa	\square	-	\square	\square
ab	\square	-	\square	\square
baa	+	+	\square	\square
bab	-	+	\square	\square
bb	-	+	+	\square



It turns out this table can be made closed and distinct – which indeed means the remaining transitions can be filled in. Here is the filled-in table and the resulting DFA:

	ε	a	aa	baa
ε	\boxplus	-	\boxplus	+
a	-	\boxplus	-	\boxplus
b	-	+	+	+
ba	+	+	+	\boxplus
aa	\boxplus	-	\boxplus	\boxplus
ab	\boxplus	-	\boxplus	\boxplus
baa	+	+	\boxplus	\boxplus
bab	-	+	\boxplus	\boxplus
bb	-	+	+	\boxplus



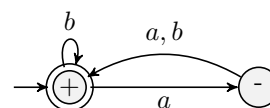
4.1 L^*_{\square} Example

We illustrate the L^*_{\square} algorithm (from Figure 5) with a teacher that starts with the following data: $L_+ = \{\varepsilon, aa, ab\}$ and $L_- = \{a, bb, abb\}$. We begin with the following table on the left as the only item in the worklist.

	ε
ε	+
a	-
b	\square

	ε
ε	+
a	-
ab	+
aa	+
b	\square

	ε
ε	+
a	-
ab	+
aa	+
b	\boxplus

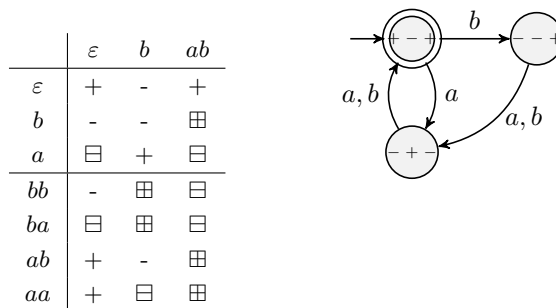


Next, we pop the table on the left off of the worklist and send it to the solver. We can see that even without filling in the blanks that the solver will return UNSAT (observe row(a) cannot possibly match row(ϵ)). So we add a to S and b to S and append the two new tables to the worklist (which was empty). We pop the middle table above off of the worklist and send it to the SMT solver. The solver finds that filling the single blank with a $+$ satisfies the constraints, resulting in the table on the right. We conjecture the corresponding machine on the right and get counterexample bb . We then add the suffixes of bb to E , and enqueue the horizontally extended (S, E, T) to the head the worklist. Thus we again pop the updated table and attempt to fill it in:

	ϵ	b	bb
ϵ	+	□	+
a	-	+	-
ab	+	-	□
aa	+	□	□
b	□	-	□

	ϵ	b	ab
ϵ	+	-	+
b	-	-	□
a	□	+	□
bb	-	□	□
ba	□	□	□
ab	+	-	□
aa	+	□	□

The table on the left is unsat. So we branch our search out by add ab , aa , and b to S on three new tables and append each of them to the worklist (which currently only holds the table with $S = \{\epsilon, b\}$). For brevity, we fast forward to the table from the last step where b was added, shown above on the right. We take the table off the worklist and the solver successfully fills in the blanks. We then conjecture the machine that corresponds to this table and this is correct on all data in L_+, L_- so the algorithm returns it.



The reader may have noticed that several times we naively added all row labels from the lower part of the table, when we could have localized the problem to a single row causing the unsat (we skipped ahead to these tables). This intuition leads to an optimization based on unsatisfiable cores, which we describe in Section 8.1.

5 Correctness of L_{\square}^*

In this section, we show that the Learner L_{\square}^* we introduced in the previous section is correct: the algorithm terminates with the minimum size automaton compatible with L_+, L_- (Theorems 10 and 11). We first show that the search proceeds monotonically in the size of S (and therefore in the size of automata).

► **Lemma 6** (Size of work lists). *The worklists generated in Figure 5 can contain tables of at most 2 sizes, n and $n + 1$. When both are present, all the tables of size n are at the front.*

Proof. The initial worklist is $[(\{\varepsilon\}, \{\varepsilon\})]$. So there is one size, $|S| = 1$, and claim is trivially satisfied. As we enter the cycle with a worklist of size n in step (2) note that we add to the worklist in two places. In step 2(a), we add a list of items of size $n + 1$ so the property is maintained in both cases (it may be that the item we popped was the last one of size n , in which case the list is now a single size). In step 2(b), we process the counterexample and add the table back to the front of the worklist – since we do not modify S this is an item of size n and the property is maintained. ◀

The following definition expresses the notion that an observation table is “on the path” to identifying some target DFA, which will be useful shortly:

► **Definition 7** (Compatible Observation Table). *An observation table (S, E, T) is compatible with a DFA $M = (Q, \Sigma, \delta, q_0, F)$ if:*

1. *The function $q: S \rightarrow Q$ defined by $q(s) = \hat{\delta}(q_0, s)$ is injective,*
2. *The blanks can be filled in by M to make the rows of S distinct, and*
3. *For each s such that $T(s) \neq \square$, then $T(s) = +$ if and only if $s \in L(M)$. (i.e. M agrees with the non-blank entries of (S, E, T)).*

In other words, a table which is compatible with a target DFA can be extended to find the target DFA by adding zero or more strings to S while maintaining the invariant that each row corresponds to a unique state. The next lemma ensures that the worklist will always have some compatible table with a minimum DFA.

► **Lemma 8.** *Let $M = (Q, \Sigma, \delta, q_0, F)$ be a minimum-size DFA consistent with L_+, L_- . All worklists generated in the algorithm in Figure 5 contain at least one table (S, E, T) that is compatible with M .*

Proof. First observe that M has no unreachable states since it is minimal. Initially, the table $(\{\varepsilon\}, \{\varepsilon\})$ is compatible with M since (a) any function from $S = \{\varepsilon\}$ is injective and (b) there is only one row, so it is distinct. We now check that the algorithm maintains the invariant of having a compatible table in the worklist. As we enter the cycle with a worklist of length n , the only interesting case is when the compatible table (S, E, T) is the head of the list, since in all other cases it will clearly still be on the worklist at the next iteration. We proceed to analyse what happens while processing (S, E, T) :

- In Step 2(a), we know that (S, E, T) could not be made closed by filling in blanks, but by induction hypothesis that the rows of S are distinguished by M . Hence, for at least one $s \in S \cdot \Sigma - S$ we have that s accesses a new state in M (if this were not the case, then SAT instance could be satisfied using M as an oracle, since then each row in the lower part would match the row in the upper part corresponding to the state accessed). By the same argument, after adding this s to S , we know it will still be distinguishable from the other rows, because if it were not, then using M as an oracle would have satisfied the core. Hence, the table obtained by adding s to S is compatible with M .
- Step 2(b)(i) does not return, so we need not maintain the invariant.
- In Step 2(b)(ii), we do not change S , so that condition (a) of Definition 7 is immediate. Any rows which can be distinguished by M using only the suffixes in E are still distinguished by additional suffixes in E' . So (S, E', T') is compatible with M . ◀

We now prove a lemma which illustrates that the crux of the search is finding the correct prefix set, S . At this point, we will make a validity query (possibly incorrectly – but the table will be satisfiable) until we are correct. If we have a particular oracle in mind (i.e., any correct automaton), then the lemma says that if S contains accessor prefixes for the states of

the oracle (and E is large enough to distinguish states), the table will be satisfiable. To be clear, in the algorithm we of course do not use an oracle to fill in the blanks – we use an SMT solver as we do not yet know the automaton. The point is simply that the satisfiability of the SMT instance when a solution exists will be used to establish correctness.

► **Lemma 9** (Existence of a solution). *Let L_+, L_- be example sets, and let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a minimum state DFA consistent with L_+, L_- . Let (S, E, T) be an observation table that is compatible with \mathcal{A} and such that $|S| = |Q|$. Then there is a closed, distinct, complete observation table (S, E, T') where $T'(s) = T(s)$ for all $T(s) \neq \square$.*

Proof. Use the DFA \mathcal{A} as an oracle to complete the table: let $T'(s) = +$ if $s \in L(\mathcal{A})$, otherwise $T'(s) = -$. Since \mathcal{A} is consistent with the datasets, so $T(s) = T'(s)$ whenever $T(s) \neq \square$.

(closed) Let $s \in S, a \in \Sigma$. If $sa \in S$, we are done, so assume $sa \notin S$. Let $q = \hat{\delta}(q_0, sa)$. By hypothesis, there is a string $s' \in S$ such that $\hat{\delta}(q_0, s') = q$. Then for each $e \in E$, $T'(sae) = \hat{\delta}(q_0, sae) = \hat{\delta}(\hat{\delta}(q_0, sa), e) = \hat{\delta}(q, e) = \hat{\delta}(\hat{\delta}(q_0, s'), e) = \hat{\delta}(q_0, s'e) = T'(s'e)$. Thus $\text{row}(sa) = \text{row}(s')$.

(distinct) By the fact that (S, E, T) is compatible with \mathcal{A} , we have that using \mathcal{A} to fill in the blanks of T distinguishes all of the rows labeled by S . ◀

► **Theorem 10** (Partial Correctness). *If the algorithm terminates, then it returns a DFA of minimal size consistent with L_+, L_- .*

Proof. Let M be any minimal DFA for L_+, L_- . Then by Lemma 8, at each point there is an observation table on the worklist that is compatible with M . Assume the algorithm eventually terminates. In the worst case, it is when we find the table (S, E, T) with $|S| = n$, for n the number of states of M , (and potentially with $E = \text{suffixes}(L_+ \cup L_-)$). By Lemma 6, we traverse the worklist in nondecreasing order. Hence, if we terminated earlier, it was by conjecturing a correct machine smaller than n , which is a contradiction. ◀

All that remains to be shown is that we make progress towards this solution and indeed the algorithm terminates.

► **Theorem 11** (Termination). *The L_{\square}^* algorithm terminates.*

Proof. Let M be a minimal DFA for L_+, L_- , and let (S, E, T) be a table in the worklist compatible with M guaranteed to exist by Lemma 8. Step 2(a) makes progress by increasing the size of S . Moreover, since there are finitely many automata M' whose size is less than or equal to M , there are only finitely many counterexamples we can get in Step 3(b) (because we never again misclassify an example we have seen). Each time we reach the compatible table, we enqueue a table with larger size by 1. When, at latest, we reach the compatible table whose size matches M , then we may need many, but only finitely many more validity queries before we are correct and terminate. ◀

6 Weakening the Teacher: iMAT with Distinguish

Next, we explore another kind of incomplete teacher that is not required to implement Validity queries, but only a weaker set of Distinguish queries. Two strings $s_1, s_2 \in \Sigma^*$ are said to be *distinguished* if there exists $e \in \Sigma^*$ such that,

- $s_1e \in L_+$ and $s_2e \notin L_+$ (or vice versa), or
- $s_1e \in L_-$ and $s_2e \notin L_-$ (or vice versa).

21:14 Automata Learning with an Incomplete Teacher

Membership: As before, the Learner sends a string u and the Teacher responds with “+” if $u \in L_+$, “-” if $u \in L_-$, and “ \square ” otherwise.

Distinguish: The learner sends two strings s_1 and s_2 , together with a finite “exclusion set” E , and the teacher responds “yes” with a suffix $e \notin E$ that distinguishes these two strings or “no” if it cannot distinguish the strings.

It should be clear that Distinguish queries are less powerful than Validity queries, as they only concern a pair of strings and not the full language. However, it turns out that when L_+ and L_- are finite, we can adapt the learner to still have a terminating procedure to learn a correct automaton. We prove these results in Section 7.

iMAT with Distinguish has close ties to foundational concepts in formal languages. In particular, the distinguish query is closely related to the Myhill-Nerode equivalence classes \equiv_L of the target language L .

► **Lemma 12.** *Given strings $s_1, s_2 \in \Sigma^*$ with distinguish \emptyset $s_1 s_2 = e$ for some $e \in \Sigma^*$, then there exists some language L that separates L_+ and L_- such that $s_1 \not\equiv_L s_2$. Moreover if, in addition, the result of membership queries to s_1e and s_2e are not \square , then for every L separating L_+ and L_- , we have that $s_1 \not\equiv_L s_2$.*

Proof. For the second claim, suppose $T(s_1e) \neq T(s_2e)$ and both are not \square . Without loss of generality, $T(s_1e) = +$ and $T(s_2e) = -$. Let L be any language separating L_+ and L_- . Clearly $s_1e \in L$ and $s_2e \notin L$. That $s_1 \not\equiv_L s_2$ follows from the definition of the Myhill-Nerode equivalence. Now for the first part, assume $T(s_1e) = -$ and $T(s_2e) = \square$ (the other cases are similar), and assume L is a regular language that separates L_+ and L_- . Then $L' = L \cup \{s_2e\}$ is also regular and separates L_+ and L_- . Moreover, for this L we have $s_1 \not\equiv_L s_2$. ◀

Although iMAT with Distinguish is guaranteed to terminate only when L_+ and L_- are finite, we can show that arbitrary problem instances can be modeled using finite instances. Hence, finite example sets are in a sense complete. More formally, suppose we are given possibly infinite sets of positive and negative examples L_+ and L_- . Then we can find finite subsets of L_+ and L_- that are sufficient for the Learner to recover the minimum-size DFA compatible with L_+ and L_- .

► **Theorem 13.** *Let $L_+, L_- \subseteq \Sigma^*$ be infinite, disjoint example sets (not necessarily regular), and let M be a minimum-size DFA that separates L_+ and L_- . Then there are finite subsets L_+, L_- such that M is also a minimum-size DFA separating L_+, L_- .*

Proof. Consider the set \mathcal{M} of all DFAs over Σ that have strictly fewer states than M . (Observe that \mathcal{M} is potentially very large, but finite). By the minimality of M on L_+, L_- , each $M' \in \mathcal{M}$ misclassifies a string from either L_+ or L_- . So we can define a function $c: \mathcal{M} \rightarrow \Sigma^*$ that maps each M' to one such counterexample string. Then the set $S = \{s \mid s = c(M'), M' \in \mathcal{M}\}$ is finite (in fact, $|S| \leq |\mathcal{M}|$). Moreover, we have by construction that for $S_+ = L_+ \cap S$ and $S_- = L_- \cap S$ that each $M' \in \mathcal{M}$ fails to separate S_+, S_- by misclassifying $c(M')$, so that M is a minimum-size separating automaton for S_+, S_- as needed. ◀

7 Correctness of L_{\square}^* with Distinguish

We now show how to adapt the L_{\square}^* to the situation that we only have an iMAT with Distinguish, not Validity. The idea is that we perform L_{\square}^* as before, but when we have a hypothesis machine, we do a series of distinguish queries instead of a validity query. Intuitively, we will ask a distinguish query for each pair of rows which are “made the same” by the hypothesis:

► **Definition 14** (Distinguish queries for a hypothesis). *Given a closed, distinct, complete table, (S, E, T) and associated DFA M , we say the distinguish queries for hypothesis M are:*

distinguish E s' s

for each $s' \in S \cdot \Sigma - S$ and for the unique s such that $\text{row}(s) = \text{row}(s')$.

Observe these distinguish queries are well-defined because each $s' \in S \cdot \Sigma - S$ is guaranteed to have a unique matching row in S .

The first theorem shows that if all of the distinguish queries for a hypothesis return None, then the hypothesis is correct.

► **Theorem 15** (Validity using Distinguish). *Suppose a learner has a hypothesis DFA $M = (Q, \Sigma, \delta, q_0, F)$ constructed from a closed, distinct, complete observation table (S, E, T) . Suppose also that the teacher returns None for all of the distinguish queries for M . Then $L_+ \subseteq L(M)$ and $L(M) \cap L_- = \emptyset$.*

Proof. We show the contrapositive. Suppose there is a counterexample string $c \in L_+$ such that $c \notin L(M)$ (without loss of generality; the other case is similar). It suffices to show that there is a distinguish query that is not None. First of all c cannot be in T since we inherit from L^* that hypotheses are correct on all evidence already considered. So we must have $c = s'_0 v_0$ for a prefix $s'_0 \in S \cdot \Sigma - S$, since $S \cdot \Sigma - S$ includes exactly one prefix of each word not in $S\Sigma$. By closedness of the table, there is an $s_0 \in S$ such that $\text{row}(s'_0) = \text{row}(s_0)$. Consider distinguish E s_0 s'_0 . If this query returns a suffix, we are done, so assume it is None. In particular, this means that $s_0 v_0 \in L_+$, since otherwise v_0 would be a response to the query. Moreover, since $\text{row}(s_0) = \text{row}(s'_0)$ we must have that $\hat{\delta}(\text{row}(s_0), v_0) = \hat{\delta}(\text{row}(s'_0), v_0)$, implying that $s_0 v_0 \notin L(M)$ and $s_0 v_0$ is also a counterexample. So, as before, there must be an $s'_1 \in S \cdot \Sigma - S$ and suffix v_1 such that $s_0 v_0 = s'_1 v_1$. Crucially, s_0 (as $s_0 \in S$) is a proper prefix of s'_1 , making v_1 a proper suffix of v_0 . Since, again, there must be an $s_1 \in S$ for which $\text{row}(s_1) = \text{row}(s'_1)$ We proceed by considering the query distinguish E s_1 s'_1 and applying the same reasoning, except that we have made progress because v_1 is a suffix of v_0 . Clearly we will see a distinguishing suffix after a finite number of these iterations: in particular by the time that $v_i = \varepsilon$, then $s'_i v_i = s'_i \in S \cdot \Sigma - S$. But that means s'_i is a string we have already seen and is also a counterexample (i.e., $s'_i \in L_+$ but rejected), which is impossible by the construction of the table (which means the distinguishing suffix must be at latest the previous round). ◀

We modify L_{\square}^* as follows:

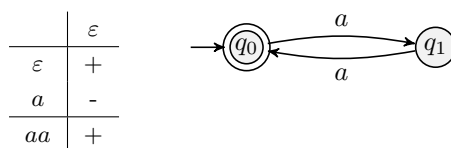
1. Run L_{\square}^* until ready to make a validity query for hypothesis H .
2. In place of the validity query, make the distinguish queries for H .
 - a. If they are all None, stop and return H .
 - b. Otherwise, add the first suffix returned by the teacher to E , and return to Step 1.

► **Corollary 16.** *If the modified L_{\square}^* terminates, the learned machine is correct.*

Proof. This follows immediately from Theorem 15. ◀

Of course, this modified procedure is not guaranteed to terminate in general. For some sparse, infinite L_+ and L_- , it is possible for the teacher to have infinitely many suffixes that the learner must consider without making progress. For example, if we have $L_+ = (aa)^*$ and $L_- = (aaaa)^*a$, the learner will soon reach:

21:16 Automata Learning with an Incomplete Teacher



At this point, the Learner will ask the query distinguish $E \varepsilon aa$ and the trouble is that the teacher has an infinite set of suffixes to give in response to distinguish $E \varepsilon aa$, namely those strings in the set $(aa)^*a$. As a result, the procedure above will not terminate in this case. We are unaware of a query weaker than Validity that still guarantees termination – settling this is an interesting question for future work.

We do have, however, as a special case, guaranteed termination for finite example sets.

► **Corollary 17.** *Let L_+ and L_- be finite, disjoint sets of strings. A learner can still learn a correct automaton for L_+, L_- with only membership and distinguish queries.*

Proof. Run L_{\square}^* modified for distinguish. Because of Theorem 15, we need only show that it terminates. Each time through Step 2 that we do not terminate, the teacher returns a suffix e . But since we add e to E , the teacher can never return this suffix again, and since the examples are finite, they can only have finitely many suffixes. Termination follows. ◀

7.1 A Landscape of Teacher Queries

As we have already presented two variants of iMAT, it is natural to ask how it relates to other formulations of automata learning problems. In this section, we summarize some known results for a variety of queries provided by a Teacher including:

Query name	Type
membership	$\Sigma^* \rightarrow 2$
equivalence	$\text{DFA} \rightarrow \Sigma^* + 1$
membership $_{\square}$	$\Sigma^* \rightarrow \{+, -, \square\}$
distinguish	$\mathcal{P}_{\text{fin}}(\Sigma^*) \times \Sigma^* \times \Sigma^* \rightarrow \{+, -, \square\}$
validity	$\text{DFA} \rightarrow \Sigma^* + 1$

Recall that though equivalence and validity have the same type, they are provided by Teachers with different capabilities: an incomplete Teacher cannot return any string not in L_+ or L_- as counterexamples as it lacks information about those strings. Hence, a Validity query returns “yes” whenever there is no evidence against the hypothesis, which is subtly different than confirming the hypothesis directly, which is what a complete teacher proves through an Equivalence query.

We distinguish two classes of learning problems. For the first class, the Teacher knows a specific regular language and will not return “unknown” in response to Membership queries. Conversely, in the second class, the teacher is incomplete and may return “unknown” in response to Membership queries.

Teacher holds concrete regular language (i.e., no blanks)

Only Membership. Suppose we run L^* , but replace each equivalence query with a loop that queries all strings looking for a counterexample. It is a consequence of the Myhill-Nerode Theorem that this process will eventually discover the correct automaton. However, the Learner will never be able to determine that it has converged. If the Learner is told the number of states, a terminating procedure is possible [1].

Only Equivalence. We can obtain a trivial terminating procedure with only the equivalence query simply by conjecturing all automata in increasing order of states. Angluin [3] showed that we cannot improve the learner to a polynomial time procedure.

Membership and Equivalence. This is Angluin’s MAT: L^* is an efficient terminating procedure for determining the correct DFA [2]. See Section 2.

Teacher is incomplete (i.e., may return blanks)

Only Membership. Just as in the non-blanks case, a learner can identify a minimum-size correct machine eventually, but cannot determine that it has done so – so there is no terminating procedure. This is Gold’s notion of “identification in the limit” [13].

Membership and Validity. Because this problem subsumes all instances of DFA inference from finite data, it is NP-Hard [15] and cannot be approximated in polynomial time (unless $P = NP$) [32]. We give a terminating procedure, L_{\square}^* , in Section 4 and prove it correct in Section 5. The complexity comments from “Only Equivalence” also hold here.

Membership and Distinguish. We consider a modification of L_{\square}^* which produces a minimum-size automaton if it terminates. We explore this in Section 6.

8 Implementation

We have built an OCaml implementation of the learner (using Validity) presented in this paper. The goal of our implementation is to provide a reusable implementation of our framework in a functional language that can be used to explore the iMAT framework, run our algorithm, and implement other algorithms that use similar primitives. Our implementation consists of two packages: (i) `nerode`, a general library for regular languages, with data structures for regular expressions, DFAs, and NFAs, as well as conversions between them, and (ii) `nerode-learn`, which implements Angluin’s L^* algorithm, as well as the iMAT variants developed in this paper. The `nerode` package is approximately 2,400 lines of code, while the `nerode-learn` package is approximately 3,000 lines of code. We use Z3 as the backend solver for checking all SMT problems.

Data Structures

Our OCaml implementation uses the following types.

- Strings are encoded in a module called `Word`, where their representation is a list of Alphabet symbols. Alphabet symbols internally are `int`.⁴

```
type Word.t = Alphabet.symbol list
```

In particular, defining modules for `Word` and `Alphabet` allowed for a modular design and allows us to support different alphabets.

- The rows of the observation table (or more precisely, the labels of the rows) are encoded as a set of Words:

```
type RowLabels.t = WordSet.t
```

- The columns of the observation table are also encoded as a set of words:

```
type ColLabels.t = WordSet.t
```

⁴ We abuse OCaml syntax slightly: `type Word.t` is not a valid type definition, but we keep the module names to show our module boundaries, and concisely differentiate all of our types `t`.

21:18 Automata Learning with an Incomplete Teacher

- Individual entries in the observation table are encoded using the following data type:

```
type entry = True | False | Blank
```

Here, the constructors include `Plus`, `Minus`, and `Blank`.

- Thus we can encode the actual lookup function of the table as a map from words to entries:

```
type TblMap.t = entry WordMap.t
```

Putting these all together, an observation table is encoded as a record s, sa, e, t where s , sa , and e are sets of strings, and $tmap$ is a map from strings to entries:

```
type Table.t = {s : RowLabels.t; sa : RowLabels.t;
               e : ColLabels.t; tmap : TblMap.t}
```

When implementing algorithms on observation tables, there is an interesting data structure choice to be made. An obvious approach is to keep the entire table explicitly as a two-dimensional array. The advantage to this approach is that the row function corresponds precisely to rows in the array. The downside is that to update the entry for a string, one has to visit each location in the table corresponding to how the string can be decomposed into prefix and suffix. To avoid this, instead of keeping the whole table, we keep S , $S \cdot \Sigma - S$, and E as sets and T as a map from strings in $(S \cup S \cdot \Sigma) \cdot E$ to table entries. Thus entries for a string are updated everywhere they appear by a single update to T . The downside is that we must generate the row function “on the fly” by repeatedly accessing T .

Algorithms

We now present our OCaml implementation of L_{\square}^* . We encode the worklist as a list of tables, which is maintained as an argument in the main recursive loop. A collection of all entries in the observation tables is also maintained globally as an argument in the loop.

```
let wl : Table.t list = ...
let entry_map : entry WordMap.t = ...
```

The main algorithm is a recursive function that searches for the smallest table that is compatible with the given positive and negative examples:

```
let rec lstar_blanks wl (e: ColLabels.t) entry_map: Dfa.t =
  let t, entry_map = List.hd wl |> Table.extend_cols entry_map e in
  match fill_blanks_smt t with
  | None →
    let new_ts, entry_map' =
      List.fold
        ~f:(fun (ts_acc, em_acc) sa →
          let new_t, em_acc' = (Table.move_up em_acc sa t) in
          new_t::ts_acc, em_acc')
        ~init:([], entry_map) (Table.lower_rows t) in
    lstar_blanks (wl @ new_ts |> List.tl) e entry_map'
  | Some obs →
    let dfa = table_to_dfa obs in
    match conjecture dfa with
    | None → dfa
    | Some cex →
      lstar_blanks wl (ColLabels.union e (suffixes cex)) entry_map
```

For the most part, the OCaml implementation follows the pseudocode given in Figure 5, but there are a few differences. In particular, the columns of the tables (e) in the worklist are not updated until right before we attempt to fill them in. We keep a cumulative mapping from strings to table entries (`entry_map`) in the main loop, preventing redundant membership

queries, and we also accumulate the suffixes learned from counterexamples (e) in the main loop. Intuitively, this approach is sound because it is safe to share counterexamples across tables and thus have the same column labels (e) for each (see Lemma 19 below).

Operationally, the `lstar_blanks` function proceeds as follows. First, it removes an item t from the worklist. It adds additional suffixes in e to the table t , using membership queries to determine new column entries in e that cannot be found in `entry_map`, filling in t with `Plus`, `Minus`, or `Blank`; it also updates the entry collection `entry_map` with any new queries. Next, it attempts to fill the blanks in the observation table encoded by t , calling `fill_blanks_smt`, which uses an SMT solver as discussed below. If the SMT solver cannot find values for the blanks, the algorithm extends the worklist with each new table new_t obtained by adding one row from the bottom part of the table t , using `Table.add`, and recurses. It also updates the `entry_map` with any new query results while filling in new rows. Otherwise, the blanks can be filled, and it makes a conjecture to the teacher. If the conjecture is correct, we have the minimal DFA. Otherwise, the conjecture fails, so we add the suffixes for the new counterexample to e , keep t at the head of `wl`, and recurse.

Efficient SMT Encoding

The key to achieving good performance in any solver-aided algorithm, is having an efficient encoding of observation tables. We use the theory of bitvectors, which is widely supported by SMT solvers, to express the blank-filling constraints. Specifically, given a table with k columns, we encode the rows in the table as bitvector variables of length k . We add assertions to constrain the bitvectors so they correspond to the rows in the table⁵. For example, the bitvector variable for the j th row in the table would be declared as follows,

```
(declare-const s_j (_ BitVec k))
```

and if the entry in the i th column was positive, we would add an assertion:

```
(assert (= ((_ extract i i) s_j) #b1))
```

Alternatively, if the entry in the i th column were a blank, we would declare a bitvector variable of length 1 to encode the blank,

```
(declare-const b_ij (_ BitVec 1))
```

and add the corresponding constraint:

```
(assert (= ((_ extract i i) s_j) b_ij))
```

Hence, a model of the SMT formula corresponds to a way of filling in the blanks that is consistent with the positive and negative entries in the table.

Returning to our algorithm, recall that we need the rows in the upper part of the table to be distinct, and we need the table to be closed. For distinctness, we use the built-in `distinct` function from the bitvector theory. For closedness, we generate assertions stating that each row in the bottom part of the table must be equal to some row in the upper part.

For example, given the observation table on the left:

⁵ An earlier version used SAT, with individual entries as boolean variables, but we found that using the bitvector theory performed better

21:20 Automata Learning with an Incomplete Teacher

	ε	a
ε	+	□
a	□	-
b	□	+
aa	-	□
ab	□	□
ba	+	-
bb	□	□

 \Rightarrow

	ε	a
ε	+	⊠
a	⊠	-
b	⊠	+
aa	-	⊠
ab	⊠	⊠
ba	+	-
bb	⊠	⊠

Our SMT encoding would have 7 bitvectors of size 2, one for each row, and 7 bitvectors of size 1, one for each (unique) blank:

```
(declare-const s1 (_ BitVec 2)) (declare-const b1 (_ BitVec 1))
(declare-const s2 (_ BitVec 2)) (declare-const b2 (_ BitVec 1))
(declare-const s3 (_ BitVec 2)) (declare-const b3 (_ BitVec 1))
(declare-const s4 (_ BitVec 2)) (declare-const b4 (_ BitVec 1))
(declare-const s5 (_ BitVec 2)) (declare-const b5 (_ BitVec 1))
(declare-const s6 (_ BitVec 2)) (declare-const b6 (_ BitVec 1))
(declare-const s7 (_ BitVec 2)) (declare-const b7 (_ BitVec 1))
```

Note that in the table above, the blank in the first row, second column and the blank in the second row, first column must have the same value as they both encode the membership of the string a in the language. Our SMT encoding account for such constraints between blanks. Next, we add assertions to encode the entries in the table:

```
(assert(=((_ extract 1 1) s1) #b1)) (assert(=((_ extract 2 2) s1) b1))
(assert(=((_ extract 1 1) s2) b1)) (assert(=((_ extract 2 2) s2) #b0))
(assert(=((_ extract 1 1) s3) b2)) (assert(=((_ extract 2 2) s3) #b1))
(assert(=((_ extract 1 1) s4) #b0)) (assert(=((_ extract 2 2) s4) b3))
(assert(=((_ extract 1 1) s5) b4)) (assert(=((_ extract 2 2) s5) b5))
(assert(=((_ extract 1 1) s6) #b1)) (assert(=((_ extract 2 2) s6) #b0))
(assert(=((_ extract 1 1) s7) b6)) (assert(=((_ extract 2 2) s7) b7))
```

Note that the assertions reflect the fact the blanks in the first rows must be filled in with the same value, since they are both associated with the string “ a ”. We would also assert distinctness for the upper part,

```
(assert(distinct s1 s2 s3))
```

Finally, we would add assertions for closedness:

```
(assert(or (= s1 s4) (= s2 s4) (= s3 s4)))
(assert(or (= s1 s5) (= s2 s5) (= s3 s5)))
(assert(or (= s1 s6) (= s2 s6) (= s3 s6)))
(assert(or (= s1 s7) (= s2 s7) (= s3 s7)))
```

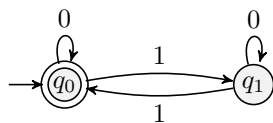
To fill in the rows in table, we can simply read off values assigned to the variables,

```
(define-fun b1 () (_ BitVec 1) #b0)
(define-fun b2 () (_ BitVec 1) #b1)
(define-fun b3 () (_ BitVec 1) #b0)
(define-fun b4 () (_ BitVec 1) #b1)
(define-fun b5 () (_ BitVec 1) #b0)
(define-fun b6 () (_ BitVec 1) #b1)
(define-fun b7 () (_ BitVec 1) #b0)
```

which corresponds to the table on the right above.

Teacher Module

Because our design relied on an abstract interface to the teacher, it was straightforward to implement the teacher in different ways. While the benchmarks we used for evaluation (Section 9) are lists of labeled examples, we also implemented a feature that allows the user to specify infinite L_+ and L_- using regular expressions. For example, we could specify $L_+ = (0101)^*$ and $L_- = 10^*$, and the system correctly identifies:



Note that the correct language has a smaller DFA than the one for L_+ !

8.1 Optimizations

We now look at several optimizations we implemented. The first one will reduce the number of tables generated in step 2(a) of Figure 5 by making use of unsat-cores; the second prevents the learner from making needlessly similar conjectures by reusing counterexamples that it gets from the teacher, and the third explores the use of a priority queue instead of a list.

Unsat-cores

When a table's blanks cannot be filled in by the SMT solver we explore the tables that result by adding each string from $S \cdot \Sigma - S$ to S . In many cases, however, we might be able to identify a row, say $s' \in S \cdot \Sigma - S$ that is already distinguished from every row in S by suffixes in E . In this case, any filling of the blanks would result in an unclosed table *because of this row*, so we can promote *only* this one, as in classic L^* .

To generalize this, we observe that the closedness assertion corresponding to the row s' above was unsatisfiable on its own. Modern SMT solvers can be configured to compute subsets of constraints that are unsatisfiable alone, called unsat cores, when returning “unsat” [26]. To justify that we are still guaranteed to find the minimal automaton boils down to extending the argument of Lemma 8:

► **Lemma 18.** *The invariant of Lemma 8 holds, even when L_{\square}^* is modified to add only tables resulting from promoting only strings in unsat core.*

Proof. The argument for Lemma 8 applies, but we need to justify that a string in the unsat core accesses a new state of M . This must be true, or else using M as an oracle would fill in the blanks to satisfy the unsat core. ◀

Suffix-set Sharing

The reader may have noticed that the L_{\square}^* Algorithm described above cleverly reuses counterexamples, accumulating the suffix set E . Hence, instead of maintaining a worklist with elements (S, E, T) , we only need a worklist that maintains prefix sets S for each individual table. For T , we maintain a cumulative mapping of strings to entries as a parameter in the main loop and simply update missing entries for columns of the popped table on each iteration. For E , we prove that sharing counterexamples is sound:

► **Lemma 19.** *Let M be a DFA, and let (S, E, T) be compatible with M and let (S, E', T') be a table such that $E \subseteq E'$. Then (S, E', T') is also compatible with M , where T' is the extension of T by membership queries for E' .*

Proof. Condition (a) of compatibility is immediate since S is unchanged. If M distinguishes the upper part rows with only the suffixes in E , they remain distinguished by E' , implying condition (b). Condition (c) holds by hypothesis. ◀

► **Corollary 20.** *The algorithm in Figure 5 is still guaranteed to produce minimal machines when modified to share a single E across the worklist.*

■ **Table 1** Performance results on established benchmarks. “Learn time” is the total time spent on an individual benchmark, while “Z3 time” is the time spent on a benchmark *within the SMT solver*. “Worklist items” are the number of tables processed from the worklist during learning.

Benchmark set	DFA Size	# Benchmarks	Mean Learn time (s)	Median Learn time (s)	Mean Z3 Time (s)	Mean worklist items
Lee, So, and Oh	2	1	0.0082	0.0082	0.0081	2.0000
	3	10	0.0234	0.0229	0.0226	5.0000
	4	9	0.0600	0.0417	0.0529	7.5556
	5	4	0.1676	0.1360	0.1401	19.0000
Oliveira and Silva	1	80	0.0056	0.0053	0.0055	1.0000
	2	15	0.0100	0.0093	0.0097	2.0667
	3	91	0.0226	0.0216	0.0213	4.0879
	4	84	0.0396	0.0338	0.0346	5.7619
	5	100	0.1237	0.0775	0.0935	8.6200
	6	105	0.3803	0.1684	0.2566	13.6381
	7	79	1.1251	0.6419	0.7583	20.0886
	8	93	10.6307	1.7830	6.1782	44.1613
	9	74	50.1672	7.7361	28.8381	96.8784
	10	25	98.0573	7.3782	65.2680	176.5200
	11	14	1498.4836	101.2503	988.9716	933.2857

Proof. Sharing E means instead of adding a counterexample to the current item’s E , traverse the worklist updating all E . By Lemma 19 adding additional strings to E does not affect compatibility, and so the invariant of Lemma 8 still holds and minimality follows. ◀

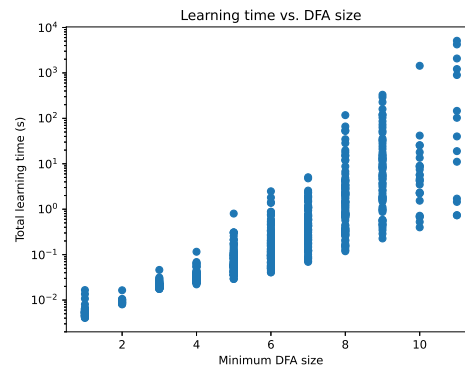
Heuristic prioritization

Our algorithm as described above uses a list that is sorted only by the size of the prefix set. By instead using a priority queue, we can apply heuristics to investigate tables which are more likely to be compatible with a minimal DFA (in the sense of Definition 7). Because we search monotonically by size, this trick can only save effort on the “last size searched,” since all smaller automata are checked first. Still, the number of prefix sets grows rapidly enough that the savings on even this last size justify the optimization.

9 Evaluation

To evaluate the performance of our implementation, we timed it on a portion of the benchmark sets created by Oliveira and Silva [29] (for their system BICA) and the benchmarks of Lee, So, and Oh⁶ [23]. We present summary data of these runs in Table 1 and Figure 7. Table 1 shows that median total learning times were consistently shorter than the mean total learning times, suggesting that, at each size, the more expensive examples are less common. The mean Z3 time column suggests the system spends around two-thirds of its running time in SMT solving at all sizes. The last column shows how the number of worklist items processed increases with DFA solution size. All benchmarks use the alphabet $\Sigma = \{0, 1\}$. We ran the experiments on a 2.10GHz Intel Xeon Silver 4216 machine with 512GB of memory.

⁶ We omit benchmark #9, which has a DFA solution of size 16.



■ **Figure 7** Performance on Oliveira and Silva benchmarks generated by automata of sizes 4 to 11.

Oliveira and Silva’s benchmark set is large: for each size from 4 states to 23 states, they produced 19 random DFAs. For each DFA, there are 5 sets of positive and negative example strings. Each example set is a set of 20 strings of length 30 produced by random walks on the generated DFA. Each problem also contains all the prefixes of those 20 strings. Using this process, they generated a total of 95 problems using each size of random DFA. Importantly, as a result of this generation process, it is very often the case that there is a smaller consistent machine than the one used in generation. In Table 1, summary data is presented for benchmarks generated from size 4 to size 11 machines, however, note that for Table 1 and Figure 8, the size shown is the size of the *learned* (i.e., minimum-size) automaton, not the one used to generate the examples.

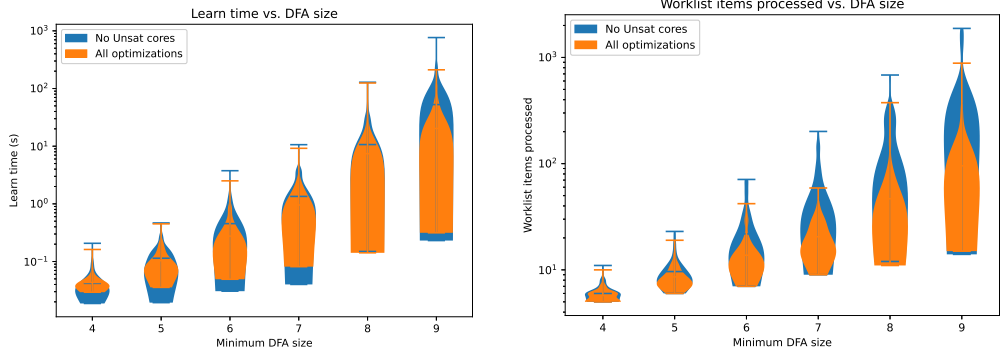
From the results of our experiments, we conclude that the scalability of our system is limited. It is practical in cases where the DFA to be learned is of size 11 or smaller. However, in its current form, learning larger DFAs is prohibitively expensive (see Section 11 for a discussion of future work in this direction). In contrast, Heule and Verwer [17] report solving all of the Oliveira and Silva benchmarks for sizes 4 to 21 “within 200 seconds per instance,” although their method requires access to all of the finite examples at the start.

9.1 Evaluation of optimizations

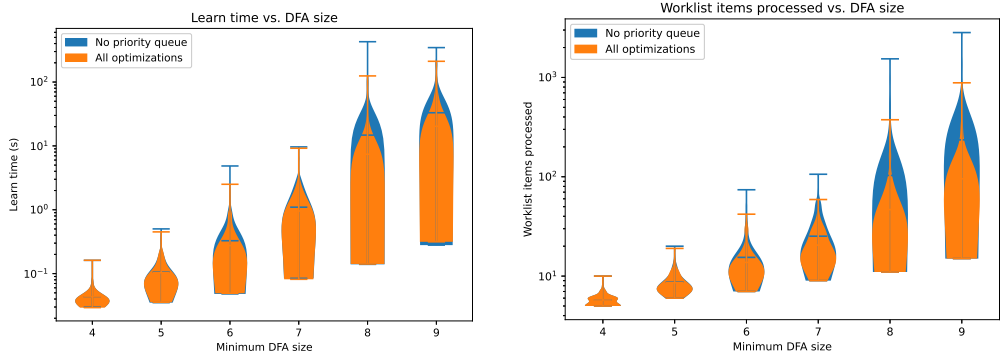
We evaluated the optimizations from Section 8.1 by conducting an ablation-style study, presented in Figure 8. The ablation study was performed in two stages. First, we ran the system on the Oliveira benchmarks (generated by sizes 4 to 9) with all optimizations turned on. Then, we ran the system on the same benchmarks a separate time for each of the three optimizations, with the optimization turned off. The results show significant time and search space savings for the unsat-cores optimization and heuristic prioritization. The “suffix-set sharing” optimization cannot affect the number of worklist items processed and turned out to result in an entirely negligible time improvement (these figures are omitted).

10 Related Work

DFA Inference from finite data. Inference of DFAs from finite data is a long standing problem and different solutions have appeared in the literature (see e.g [10] for an overview). Gold introduced the observation table and considered blank entries (“holes”) in the context of passive learning, but his algorithm does not guarantee minimality of the automaton produced [14]. Modern algorithms attack the problem from the perspective of “state



(a) Time reduction from unsat core optimization. (b) Search space reduction of unsat core optimization.



(c) Time reduction from heuristic prioritization. (d) Search space reduction of heuristic prioritization.

■ **Figure 8** Ablation study of optimizations proposed in Section 8.1. Data for “Suffix-set sharing” is omitted because it produced only negligible speedup.

merging” [30]. The main idea of these approaches is to build an automaton from the tree of all prefixes of positive examples, called the prefix tree acceptor (PTA), and then attempt to merge states, using the negative examples to validate merges. It is invariant that the positive examples are accepted, but a merge might cause a negative example to be accepted – in which case it is backtracked. Much research has gone into the selection of which merges to prioritize [21, 9]. State merging methods are much faster than the algorithm described in this paper but in general do not guarantee minimum size. A notable exception to this is the **exbar** Algorithm due to Lang [20], which gives a method for exhaustively exploring potential merges in a way that finding a minimum size DFA is guaranteed at exponential running time cost.

Inference of Separating Automata. Chen et al [8] use a similar notion of blanks in L^* (which they call “Don’t care”). The problem they solve is closely related, but it requires that the teacher start with two *regular* languages, for which they seek to find a minimal separating DFA. The algorithm they present, L^{sep} , computes the final DFA by first computing a 3-valued automaton (3DFA), which has “Don’t care” as output. They then convert this automaton to the minimum-size consistent DFA. Crucially, the interface they define with the teacher requires that the positive and negative example sets be regular languages. Thus in light of this restriction, the problem solved by L^{sep} may be seen as a special case of the iMAT setting.

DFA Inference using SAT solvers. Oliveira and Silva use constraint solvers to find a mapping from the states of a PTA to a particular size [29]. If the search fails, the size is increased until a DFA is found. Their work extends the method of Biermann and Feldman [6], who first explored such mappings from the PTA. Heule and Verwer [17] also use SAT solvers to infer DFAs from examples but their approach uses a SAT formulation closely tied to graph coloring. They first build a compatibility graph for states of the PTA, where states are compatible if their merge is not immediately ruled out by examples. The colors assigned in the coloring instance thus correspond to states in the smaller DFA, with the edges of the incompatibility graph ruling out incorrect merges.

Active learning of Network Invariants. Grinchtein, Leucker, and Piterman [16] present an algorithm that augments L^* to handle missing information with a SAT solver for inferring network invariants – i.e., in the sense of Wolper and Lovinfosse’s inductive technique verifying for large compositions of finite-state systems [35]. Their work, which improves the earlier method of Pena and Oliveira [31], extends the Angluin notions of table closedness and consistency to tables with blanks (called “weakly closed” and “weakly consistent”). The learner can proceed by performing L^* corresponding actions on tables while there are still blanks. Once the table is weakly closed and weakly consistent, they produce a series of SAT queries following Biermann and Feldman’s approach ([6], also mentioned below). The result of these SAT queries is a minimum-size automaton consistent with the examples in the table – in their scheme, they do not maintain the invariant that the minimum automaton is equal in size to the upper part of the table as in our approach, so they do not necessarily conjecture hypotheses monotonically.

In follow-on work Leucker and Neider [24] presented a framework which distills the essentials of the algorithm of [16] for learning DFAs. Their formal framework presents the teacher similar to our presentation in Section 3, but they do not highlight the solver as a first-class citizen in the framework and think of it as part of the Learner. They do give an overview of potential learners that work with inexperienced teachers. These encompass an approach without membership queries (a naive enumeration of all DFAs of increasing size) and the approach of [16] with a SAT solver, as well as the approach of [8], which we described above. They do not consider any modification similar to our iMAT with Distinguish, and they do not explore any implementation aspects or benchmarking.

11 Discussion

We have presented algorithms that solve the problem of automata inference from an incomplete teacher. The core ideas we applied to produce our algorithm are data structure agnostic and therefore a first direction for future work is whether we can adapt the work we did in this paper to recent optimizations of L^* , such as L^\sharp .

A particularly interesting optimization is the use of *discrimination trees*, an efficient replacement for observation tables [19, 18]. The first challenge will be to understand how the operations on discrimination trees can be generalized in the setting with \square ’s.

Another interesting direction would be to look at L^* variants developed for other automata models. We expect that the work in this paper applies directly to Mealy and Moore machines, with the caveat that the number of options for filling in blanks will be affected by the size of the output sets. But one could also explore how SMTs solvers could help in learning of weighted and symbolic automata, two models with interesting applications for which L^* -like algorithms were proposed [5, 4, 12].

Finally, there is another L^* adaptation, due to Bollig et al., that learns non-deterministic finite automata with access to an Angluin MAT. Another interesting direction for future work would be to investigate the adaptation of their approach to the incomplete setting.

References

- 1 Dana Angluin. A note on the number of queries needed to identify regular languages. *Inf. Control.*, 51:76–87, 1981.
- 2 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.
- 3 Dana Angluin. Negative results for equivalence queries. *Mach. Learn.*, 5(2):121–150, July 1990. doi:10.1023/A:1022692615781.
- 4 Borja Balle and Mehryar Mohri. Learning weighted automata. In Andreas Maletti, editor, *Algebraic Informatics*, pages 1–21, Cham, 2015. Springer International Publishing.
- 5 Francesco Bergadano and Stefano Varricchio. Learning behaviors of automata from multiplicity and equivalence queries. *SIAM J. Comput.*, 25(6):1268–1280, December 1996. doi:10.1137/S009753979326091X.
- 6 A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972. doi:10.1109/TC.1972.5009015.
- 7 Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, pages 487–502, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3385988.
- 8 Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating dfa’s for compositional verification. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 9 Orlando Cicchello and Stefan C. Kremer. Beyond edsm. In *Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications, ICGI ’02*, pages 37–48, Berlin, Heidelberg, 2002. Springer-Verlag.
- 10 Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010. doi:10.1017/CB09781139194655.
- 11 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.
- 12 Samuel Drews and Loris D’Antoni. Learning symbolic automata. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 173–189, 2017. doi:10.1007/978-3-662-54577-5_10.
- 13 E. Mark Gold. Language identification in the limit. *Inf. Control.*, 10:447–474, 1967.
- 14 E. Mark Gold. System identification via state characterization. *Automatica*, 8(5):621–636, September 1972. doi:10.1016/0005-1098(72)90033-7.
- 15 E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. doi:10.1016/S0019-9958(78)90562-4.

- 16 O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In *3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 483–497. Springer-Verlag, 2006.
- 17 Marijn J. H. Heule and Sicco Verwer. Exact dfa identification using sat solvers. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications*, pages 66–79, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 18 Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734, pages 307–322. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-319-11164-3_26.
- 19 Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994.
- 20 Kevin J. Lang. Faster algorithms for finding minimal consistent dfas. Technical report, NEC Research Institute, 1999.
- 21 Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference, ICGI '98*, pages 1–12, Berlin, Heidelberg, 1998. Springer-Verlag.
- 22 Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. *SIGPLAN Not.*, 49(6):542–553, June 2014. doi:10.1145/2666356.2594333.
- 23 Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016*, pages 70–80, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2993236.2993244.
- 24 M. Leucker and Daniel Neider. Learning minimal deterministic automata from inexperienced teachers. In *Leveraging Applications of Formal Methods*, 2012.
- 25 Yeting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. Transregex: Multi-modal regular expression synthesis by generate-and-repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1210–1222. IEEE, 2021.
- 26 Mark Liffiton and Karem Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40:1–33, January 2008. doi:10.1007/s10817-007-9084-z.
- 27 Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. In *COLT 1991*, 1991.
- 28 Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michał Szynwelski. Learning nominal automata. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, POPL '17, pages 613–625, 2017.
- 29 Arlindo Oliveira and J.P.M. Silva. Efficient algorithms for the inference of minimum size dfas. *Machine Learning*, 44, July 2001.
- 30 Jose Oncina and Pedro García. Inferring regular languages in polynomial update time. *World Scientific*, January 1992. doi:10.1142/9789812797902_0004.
- 31 J.M. Pena and A.L. Oliveira. A new algorithm for exact reduction of incompletely specified finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(11):1619–1632, 1999. doi:10.1109/43.806807.
- 32 Leonard Pitt and Manfred K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, January 1993. doi:10.1145/138027.138042.
- 33 Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, January 2017. doi:10.1145/2967606.

- 34 Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A New Approach for Active Automata Learning Based on Apartness. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 223–243. Springer, 2022. doi:10.1007/978-3-030-99524-9_12.
- 35 Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 68–80, 1990.
- 36 Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. Interactive program synthesis by augmented examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pages 627–648, 2020.

A Another L_{\square}^* Example

Suppose we start with the following input.

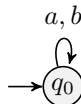
$$L_+ = \{ab, aab, bab, aaab, abab, baab, bbab\}$$

$$L_- = \{aa, ba, bb, aaa, baa, aba, bba, abb, bbb\}$$

We begin with the following table as the only item in the worklist.

	ε
ε	□
a	□
b	□

	ε
ε	⊠
a	⊠
b	⊠



We pop this table off of the worklist, and the SMT solver attempts to fill in its blanks. The solver finds that filling all the blanks with a - satisfies the constraints. We then conjecture the machine that corresponds to the filled-in table.

We get the counterexample *baab*, add its suffixes to *E*, and push the new (S, E, T) , below on the left, onto the top of worklist.

	ε	b	ab	aab	baab
ε	□	□	+	+	+
a	□	+	+	+	□
b	□	-	+	+	□

	ε	b	ab	aab	baab
ε	□	□	+	+	+
a	□	+	+	+	□
aa	-	+	+	□	□
ab	+	-	+	□	□
b	□	-	+	+	□

Now the table on the left is unsatisfiable. Hence, we add each element in $S\Sigma - S$ to *S* and add each resulting (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, a\}$ (table depicted on the right above), and the last item has $S = \{\varepsilon, b\}$. Once again, the solver fails because there is no way to fill in the blanks and maintain closedness. Once more, we add each element in $S\Sigma - S$ to *S* and add each new (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, b\}$, the next item has $S = \{\varepsilon, a, aa\}$, and the last item has $S = \{\varepsilon, a, ab\}$.

	ε	b	ab	aab	$baab$
ε	\square	\square	$+$	$+$	$+$
b	\square	$-$	$+$	$+$	\square
a	\square	$+$	$+$	$+$	\square
ba	$-$	$+$	$+$	\square	\square
bb	$-$	$-$	$+$	\square	\square

We pop the table above off of the worklist. Again the solver fails because there is no way to fill in the blanks and maintain closedness. We add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. Now the head of the worklist has $S = \{\varepsilon, a, aa\}$, the next item has $S = \{\varepsilon, a, ab\}$, the next item has $S = \{\varepsilon, b, ba\}$, and the last item has $S = \{\varepsilon, a, bb\}$.

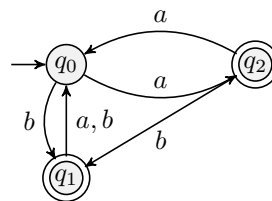
In the next step, we pop the table below on the left off of the worklist. Unsat again. We add each element in $S\Sigma - S$ to S and add each new (S, E, T) to the tail of the worklist. At the head of the list we now have the table on the right.

	ε	b	ab	aab	$baab$
ε	\square	\square	$+$	$+$	$+$
a	\square	$+$	$+$	$+$	\square
aa	$-$	$+$	$+$	\square	\square
aaa	$-$	$+$	\square	\square	\square
aab	$+$	\square	\square	\square	\square
ab	$+$	$-$	$+$	\square	\square
b	\square	$-$	$+$	$+$	\square

	ε	b	ab	aab	$baab$
ε	\square	\square	$+$	$+$	$+$
a	\square	$+$	$+$	$+$	\square
ab	$+$	$-$	$+$	\square	\square
aa	$-$	$+$	$+$	\square	\square
aba	$-$	$+$	\square	\square	\square
abb	$-$	\square	\square	\square	\square
b	\square	$-$	$+$	$+$	\square

This time SMT solver successfully fills in the blanks:

	ε	b	ab	aab	$baab$
ε	\boxplus	\boxplus	$+$	$+$	$+$
a	\boxplus	$+$	$+$	$+$	\boxplus
ab	$+$	$-$	$+$	\boxplus	\boxplus
aa	$-$	$+$	$+$	\boxplus	\boxplus
aba	$-$	$+$	\boxplus	\boxplus	\boxplus
abb	$-$	\boxplus	\boxplus	\boxplus	\boxplus
b	\boxplus	$-$	$+$	$+$	\boxplus



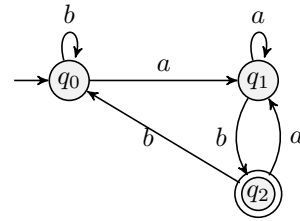
We get counterexample bbb , add its suffixes to E , and the head of the worklist becomes:

	ε	b	ab	aab	$baab$	bb	bbb
ε	\square	\square	$+$	$+$	$+$	$-$	$-$
a	\square	$+$	$+$	$+$	\square	$-$	\square
ab	$+$	$-$	$+$	\square	\square	\square	\square
aa	$-$	$+$	$+$	\square	\square	\square	\square
aba	$-$	$+$	\square	\square	\square	\square	\square
abb	$-$	\square	\square	\square	\square	\square	\square
b	\square	$-$	$+$	$+$	\square	$-$	\square

Finally the SMT solver successfully fills in the blanks:

21:30 Automata Learning with an Incomplete Teacher

	ε	b	ab	aab	$baab$	bb	bbb
ε	\square	\square	+	+	+	-	-
a	\square	+	+	+	\boxplus	-	\square
ab	+	-	+	\boxplus	\boxplus	\square	\square
aa	-	+	+	\boxplus	\boxplus	\square	\square
aba	-	+	\boxplus	\boxplus	\boxplus	\square	\square
abb	-	\square	\boxplus	\boxplus	\boxplus	\square	\square
b	\square	-	+	+	\boxplus	-	\square




The corresponding automaton is consistent with L_+ , L_- so we return it as the result.


Modular Verification of State-Based CRDTs in Separation Logic

Abel Nieto 

Aarhus University, Denmark

Arnaud Daby-Seesaram 

ENS Paris-Saclay, France

Léon Gondelman 

Aarhus University, Denmark

Amin Timany 

Aarhus University, Denmark

Lars Birkedal 

Aarhus University, Denmark

Abstract

Conflict-free Replicated Datatypes (CRDTs) are a class of distributed data structures that are highly-available and weakly consistent. The CRDT taxonomy is further divided into two subclasses: state-based and operation-based (op-based). Recent prior work showed how to use separation logic to verify convergence and functional correctness of op-based CRDTs while (a) verifying implementations (as opposed to high-level protocols), (b) giving high level specifications that abstract from low-level implementation details, and (c) providing specifications that are modular (i.e. allow client code to use the CRDT like an abstract data type). We extend this separation logic approach to verification of CRDTs to handle state-based CRDTs, while respecting the desiderata (a)–(c). The key idea is to track the state of a CRDT as a function of the set of operations that produced that state. Using the observation that state-based CRDTs are automatically causally-consistent, we obtain CRDT specifications that are agnostic to whether a CRDT is state- or op-based. When taken together with prior work, our technique thus provides a unified approach to specification and verification of op- and state-based CRDTs. We have tested our approach by verifying StateLib, a library for building state-based CRDTs. Using StateLib, we have further verified convergence and functional correctness of multiple example CRDTs from the literature. Our proofs are written in the Aneris distributed separation logic and are mechanized in Coq.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Distributed algorithms; Theory of computation → Separation logic

Keywords and phrases separation logic, distributed systems, CRDT, replicated data type, formal verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.22

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*:
<https://doi.org/10.4230/DARTS.9.2.15>

Funding This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

1 Introduction

Conflict-free Replicated Data Types (CRDTs) are a class of distributed data structures that trade off strong consistency in favour of high availability. That is, local updates are not blocked by inter-replica synchronization; instead, they are immediately applied locally, and



© Abel Nieto, Arnaud Daby-Seesaram, Léon Gondelman, Amin Timany, and Lars Birkedal;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 22; pp. 22:1–22:27



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



then propagated to other replicas. Because of the lack of synchronization, CRDTs need a mechanism for resolving conflicting updates: a typical strategy is to make all updates commutative, so that they can be applied in any order.

There are two main implementation strategies for CRDTs, differing in how updates are propagated. *Operation-based* (op-based) CRDTs propagate local updates by reifying updates as operations, which are then transmitted to other nodes. Once these (now remote) operations are received by other replicas, they can be applied to their local states so they can “catch up”. By contrast, in *state-based* CRDTs, an update is first applied to the local state, and then the *state* is propagated to other replicas. This is achieved by letting the state be an element of a join-semilattice, constraining updates to be monotonic, and combining local and remote states via the lattice’s join operator. The choice of implementation style for a CRDT (op vs state-based) incurs several tradeoffs. Op-based based CRDTs are conceptually simpler, but make assumptions about the underlying delivery mechanism (e.g. at most once delivery). By contrast, the state-based approach can easily cope with duplicates and messages delivered out of order, because the merge operation (modelled with joins) is idempotent, associative, and commutative. On other hand, not only must the datatype semantics be encoded via joins, but also sending the entire state across the network might be inefficient.

Figure 1 shows a *grow-only counter* (g-counter) CRDT implemented in both styles. A g-counter is a datatype with two operations: it can be read and it can be incremented by a non-negative number. The op-based implementation defines the counter’s initial state (0), an `effect` function that adds the value we are incrementing by to the counter’s current state, and a `read` function that is just the identity. An *event* is a tuple containing an operation (the value to increment by) plus metadata, including the replica id where the operation originated. The state-based g-counter is more complex. The counter’s state is kept as a list of integers tracking each replica’s contribution to the counter’s value. The initial state is the list of all zeroes with size `numRep`, the number of replicas. A `mutator` function takes the current state and the increment value, and returns the updated list where the right entry, as determined by the operation’s origin, was incremented. The `merge` function takes two states and computes their join in the underlying lattice. For the g-counter, we take the pointwise maximum of the two lists. Finally, to read the value of the g-counter we just sum all entries in the state list. These purely functional implementations capture the g-counter’s core logic, but do not show how events are propagated between replicas.

The standard consistency model for CRDTs is *Strong Eventual Consistency* (SEC). SEC can, in turn, be divided into two sub-properties: *convergence* (two replicas that have processed the same set of updates must be in the same state) and *eventual delivery* (any update sent by a replica will eventually be delivered to all other replicas).

Additionally, the guarantees of SEC are sometimes strengthened to imply *causal consistency* [2], meaning that the causal order of updates is respected. In other words, given updates u and w , if u happened before w at a replica [12], then u must be processed before w at all other replicas. Causal consistency captures programmer’s intuitions on how the order of operations should be preserved; for example, it implies that reads are monotonic: a read always returns data that is “fresher” than what previous reads have returned.

1.1 Denotational Specifications

Specifying CRDTs is tricky because of their replicated nature and relaxed consistency model. Some works adopt SEC as the correctness criteria and do not provide functional correctness specifications of CRDTs [18, 7, 17]. Eventual consistency is a key correctness property, but this approach has at least two (related) shortcomings. First, given, e.g., a g-counter

```

(* op-based g-counter *)
let init = 0

let effect st e =
  let (op, _, _) = e in
  st + op

let read st = st

(* state-based g-counter *)
let init = List.init numRep (fun _ → 0)

let mutator st e =
  let (op, src) = e in
  List.mapi (fun i c →
    if i = src then c + op else c)

let merge st1 st2 =
  let max = fun p →
    Int.max (fst p) (snd p) in
  List.map max (List.combine st1 st2)

let read = List.fold_left (+) 0

```

■ **Figure 1** Op-based and state-based OCaml implementations of a grow-only counter. `numRep` is the number of replicas.

implementation, proving SEC shows that different replicas eventually converge, but does not tell us what they converge to. This means we cannot rule out bugs such as subtracting instead of adding in the definition of `effect` in Figure 1. Another problem is that by focusing on SEC we cannot abstract away from implementation details. For example, Figure 1 shows two g-counter implementations that use different techniques, but we should be able to reason about a g-counter as an *abstract data type* [15], without worrying about whether it is op-based or state-based.

Burckhardt et al. [3] were the first to give functional correctness specifications of CRDTs. Their key observation is that just like a sequential data type (e.g., a queue) can be specified as a partial function from a list of operations to a (final) state, a replicated data type can be specified as a partial function from a *set of events* to a state. As in Figure 1, an *event* contains an operation plus additional metadata, including the id of the replica that created the operation and a timestamp. The timestamps induce a partial or total order on the set of events and, furthermore, that order is consistent with causality.¹

We call this partial function from sets of events to the CRDT’s state after processing the events a *denotation*.² For example, the denotation for a g-counter is $\llbracket s \rrbracket = \sum_{e \in s} e.o$, where s is a set of events and $e.o$ extracts e ’s operation (the value to increment by). In this particular case we do not use the event metadata to specify the g-counter, but we do so for other, more complex, CRDTs where all operations do not naturally commute. Note that any CRDT specification that uses denotations trivially satisfies the convergence part of SEC, because $\llbracket \cdot \rrbracket$ is insensitive to the order in which events arrive at different replicas: if they have received the same set of events, then their states will be the same. Also notice that denotations are by nature closer to the informal op-based specification in Figure 1 than to the state-based one. This is because op-based CRDTs are framed in terms of individual operations.

1.2 Verifying with Denotations

The papers by Burckhardt et al. and Leijnse et al. [3, 13] are concerned with specifying CRDTs, but there is still a need for a mechanism that ties the high-level specifications, given in terms of denotations, to executable code written using features of a modern programming language: e.g., mutation, node-local concurrency, and higher-order functions. The recent

¹ In Burckhardt et al. [3] a *visibility* relation is used to order events, instead of a timestamp.

² The term is due to Leijnse et al. [13], who recast Burckhardt et al.’s formalism in a style more amenable to specifying CRDT combinators.

work of Nieto et al. [20] connects denotations to low-level CRDT implementations using the Aneris distributed separation logic [11]. Specifically, they show how to build separation logic propositions that track the local state of a CRDT, where, e.g., the return value of a read is then given by a denotation of the local state. Nieto et al. demonstrate their approach by verifying a library for building op-based CRDTs: the library user (the CRDT implementer) instantiates the library with a purely-functional implementation of the CRDT (similar to the op-based example in Figure 1), and obtain in return a replicated data type. The library handles network operations, concurrency control, and mutation of local state. Nieto et al. exclusively reason about operation-based CRDTs. As future work, the authors include a high-level sketch of how their techniques might be adapted to the state-based setting.

There is then, to the best of our knowledge, an unexplored gap in the literature for verifying functional correctness of *state*-based CRDTs using modular specifications.

Related to the last point, existing approaches to verifying CRDTs target either op-based [7, 16, 14, 17, 20] or state-based [23, 18, 24] CRDTs, but never both kinds. This is important because it means that it is not possible to give the *same* specification to two implementations of the same replicated data type, where each uses a different implementation strategy (as in Figure 1). Having specifications that hide away implementation details is something we take for granted for sequential data types (e.g. a set abstract data type can be implemented both via a linked list and a hash table, but both implementations can be given the same specification). It would be useful to have the same hold for CRDTs.

1.3 Contributions

We fill the gaps identified above through the following contributions:

1. We give the first modular specification of a general class of state-based CRDTs. Our specifications are given in the Aneris distributed separation logic and our proofs are mechanized in Coq.
2. Furthermore, when taken together with Nieto et al. [20], our work provides a *unified* framework for the specification and verification of *both* kinds of CRDTs. This is because our specifications of state-based CRDTs are compatible with Nieto et al.’s specifications of op-based CRDTs. We emphasize this point by re-verifying the example client program in Nieto et al. that uses a positive-negative counter CRDT, except we swap their op-based counter by a state-based equivalent. Crucially, the client’s safety proof remains virtually unchanged,³ showing that it is possible to specify CRDTs while hiding their implementation strategy.
3. We give the first formal proof that state-based CRDTs are causally-consistent.
4. We evaluate our approach by verifying a set of example CRDTs from the literature. The evaluation shows that our techniques can handle a variety of CRDTs, including counters, sets, and higher-order combinators.

2 Aneris Primer

Aneris [11] is a distributed separation logic built on top of the Iris program logic framework [9]. Aneris is designed to reason about safety properties of distributed systems written in Aneris-Lang, which can be thought of as a subset of OCaml deeply embedded in Coq. This subset

³ Modulo some manual editing of Nieto et al.’s proof development, which could be further eliminated with additional Coq engineering work that refactors some typeclasses

includes support for higher-order functions, mutable state, node-local concurrency (including the ability to fork new threads dynamically), as well as expressions for sending and receiving messages over UDP-style sockets. The operational semantics models an unreliable network: once sent, messages can be dropped, re-ordered, arbitrarily-delayed, and duplicated.

Figure 2 shows the fragment of Aneris most relevant to this paper. First, notice that the logic includes the usual connectives of a higher-order logic. The separation logic connectives include the separating conjunction $P * Q$, indicating ownership of a resource that can be split into two parts, one satisfying P and the other satisfying Q . The magic wand $P \multimap Q$ denotes resources that, when combined with a resource satisfying P then together satisfy Q . The points-to proposition $\ell \mapsto_{ip} v$ grants exclusive ownership of memory location ℓ on the node with IP address ip alongside the knowledge that value v is stored in ℓ . In other words, only the owner of this resource is allowed to read or modify ℓ 's contents. As usual, the Hoare triple $\{P\} \langle ip; e \rangle \{x. Q\}$ is a partial correctness assertion for expression e running on the node with IP address ip . Notice that in the postcondition we bind the return value of e to x , whose scope extends over Q .

Aneris inherits from Iris a notion of *invariant*. An invariant $\boxed{P}^{\mathcal{N}}$ (\mathcal{N} being the name – see below), once established at a point in a proof, asserts that the proposition P hold throughout the execution of the program from that point on and is respected by all threads and nodes. This is enforced by the program logic and is reflected in the invariant *opening* rule. The invariant opening rule allows invariants to be accessed around atomic expressions. That is, it allows us to assume that the invariant holds before the atomic step and enforces that after the atomic step executes, the invariant needs to be *closed* again, meaning that we have to show that it holds again after the execution of the atomic step. The notation $\boxed{P}^{\mathcal{N}}$ says that P is an invariant with *namespace* \mathcal{N} . One can think of \mathcal{N} as an identifier for the invariant that helps the logic keep track of which invariants are open at any given point in the proof: this is important because an invariant that is currently open cannot be re-opened.

The points-to proposition $\ell \mapsto_{ip} v$ is but one example of the kind of *resources* that one can assert ownership over. In fact, the user of the logic can define new kinds of resources by creating *partial-commutative monoids* (PCMs): monoids where the product is commutative and partial. Given a PCM \mathbb{M} and $a \in \mathbb{M}$ the proposition \boxed{a}^{γ} asserts ownership of *ghost state* a . Here, γ is the *ghost name* under which a is stored. Crucially, $\boxed{a}^{\gamma} * \boxed{b}^{\gamma}$ is equivalent to the monoid product $\boxed{a \cdot b}^{\gamma}$. The logic guarantees that the product of all resources stored under the same ghost name is well-defined: hence by choosing appropriate monoids we can tweak the properties of ghost state.

The proposition $\Box P$, read *persistently* P , tells us that P holds and it does not assert ownership of any exclusive resources. We say that P is a persistent proposition if $P \vdash \Box P$. Persistent propositions are duplicable in the sense that $\Box P \vdash \Box P * \Box P$. Finally, the *update* modality $\mathcal{E}_1 \boxRightarrow^{\mathcal{E}_2} P$ says that P holds after updating (allocating or modifying) resources; furthermore, we can assume that all invariants in the set \mathcal{E}_1 hold when establishing P but must also (re)establish all invariants in the set \mathcal{E}_2 . The notation $\boxRightarrow_{\mathcal{E}}$ is shorthand for $\mathcal{E} \boxRightarrow^{\mathcal{E}}$.

3 Main Ideas

The main idea of this paper is that, from a user's perspective, whether a CRDT is op-based or state-based is an implementation detail, and one that ought not affect the data structure's specification. To capture this idea formally we reach for two tools: separation logic propositions for tracking the global and local states of the CRDT as a function of sets of events (user operations), and high-level specifications for the CRDT based on denotations.

$P, Q \in iProp ::= \text{True} \mid \text{False} \mid P \wedge Q \mid P \Rightarrow Q \mid P \vee Q \mid \forall x. P \mid \exists x. P \mid \dots$	higher-order logic
$\mid P * Q \mid P \multimap Q \mid \ell \mapsto_{ip} v \mid \{P\} \langle ip; e \rangle \{x. Q\}$	separation logic
$\mid \boxed{P}^{\mathcal{N}} \mid \overline{a_i}^{\gamma}$	invariants and resources
$\mid \Box P \mid \varepsilon_1 \multimap \varepsilon_2$	modalities

■ **Figure 2** Aneris fragment adapted from Nieto et al. [20].

To track a CRDT’s state we construct propositions $\text{GlobSt}(s_g)$ and $\text{LocSt}(i, s_{own}, s_{for})$ for global and local states, respectively, as well as a global invariant GlobInv . Ownership of the global state tells us that s_g is *exactly* the set of all events issued by any replica. An event is a triple (**op**, **source**, **time**) where **op** is the user-provided operation (e.g. `inc(10)` for a g-counter), **source** is the id of the replica that issued the operation (ids are just natural numbers), and **time** is a logical timestamp that allows us to order events according to the usual *happens-before* relation. Ownership of the local state $\text{LocSt}(i, s_{own}, s_{for})$ tells us that replica i has issued *exactly* the events in s_{own} , and that it has received *at least* the events in s_{for} , which all originate outside of i (we only get an approximation of the events from outside of i because in between two user interactions with the CRDT at replica i , new merge operations might have taken place).

Once the above propositions have been defined, we prove a comprehensive suite of “resource lemmas” that allow a client to reason about the state of the CRDT. For example, we prove that if a user knows both $\text{GlobSt}(s_g)$ and $\text{LocSt}(i, s_{own}, s_{for})$, and they can find events e and e' where $e \in s_{own} \cup s_{for}$ (e is in the local state), $e' \in s_g$ (e' is in the global state) and e' happens before e , then e' must have been received at replica i as well: $e' \in s_{own} \cup s_{for}$. This is an encoding of causality in separation logic [8, 20]. We do not claim this formulation as a contribution; instead, our contribution is defining the above predicates in such a way that they can track state-based CRDTs, and then proving existing lemma statements for our definitions. In effect, we have re-implemented an existing interface. This is crucial because it means that from a client’s perspective it does not matter what kind of CRDT (state-based or op-based) they are working with: they all satisfy the same laws.

Technical Challenges

Resource lemmas like causality hold only in the presence of the global invariant GlobInv . This invariant guarantees that at all times the state of the system is *valid*. The system state is a tuple \vec{s}_i of the local state at every replica. Defining a notion of validity suitable for state-based CRDTs is one of the tricky technical parts of our proof. Two complications arise: how to represent local states, and how to represent time.

Defining local state as an element of the lattice over which the CRDT is implemented does not work, because we lose track of the individual events. For example, we might remember that the state of the counter is [4, 5], but not that it resulted from three events: two increments of 2 each at replica 0, and one increment of 5 at replica 1. Remembering events is needed to show the resource lemmas. We therefore represent local states as sets of events, but now we need a way to link these sets of events to the lattice element that is computed at runtime. We do this with denotations, which are functions from sets of events to lattice elements. In our proof, a local invariant ensures that, for every replica i , if s is the set of tracked events for i and the runtime CRDT state at i is st , we must have $\llbracket s \rrbracket = st$. The definition of denotation for a state-based CRDT must then satisfy a number of *coherence* properties with respect to the underlying lattice: for example, we require that, under certain

conditions, if $\llbracket s \rrbracket = st$ and $\llbracket s' \rrbracket = st'$, then $\llbracket s \cup s' \rrbracket = st \sqcup st'$. In other words, our proof tracks the logical state with a free lattice of events, while the implementation computes using the CRDT-specific lattice. The denotation is the homomorphism between the two.

For lemmas like causality we also need to be able to compare events according to time. Prior work on op-based CRDTs implements a causal broadcast library that tags each event with a (physical) *vector clock*⁴ that serves as a timestamp [20]. In our setting, since state-based CRDTs do not assume causal broadcast, we use a purely logical notion of time. Given an event e , its timestamp is taken to be the set of events that e causally depends on. This set is computed at the point e is created: if e was issued at replica i where the local state is an event set s , then e causally depends on every element of s . This works because we can show that local states are always *dependency-closed*: if $e' \in s$ and e_d is a dependency of e' , then $e_d \in s$ as well. Our definition of logical time allows us to give the first formal proof that state-based CRDTs are causally-consistent.

Verified Examples

With the above in place we turn to the verification of different state-based CRDTs. We implemented and verified five CRDTs from the literature [21]: a grow-only counter, a positive-negative counter, an add-only set, a combinator for products of CRDTs, and a combinator for maps from strings to an underlying CRDT type. We implemented these examples in two steps: first, we implemented a STATELIB library that factors out all the common elements in different examples: network calls, merging of remote states, and concurrency control. The library takes as input a purely-functional implementation of a state-based CRDT, in the form of a triple $(\text{init_st}, \text{mutator}, \text{merge})$, where init_st is the CRDT's initial (lattice based) state, mutator is a function that takes a state and an operation and produces the next state, and merge implements the lattice's least upper bound operation. STATELIB requires that the CRDT implementer proves the aforementioned coherence properties (e.g., that merging two states is the same as taking the denotation of the union of their corresponding event sets) about their purely-functional implementation. Given this core logic, the library returns a fully-fledged replicated data type and two functions, get_state and update , to query and update the state of the data-structure. In the second step, we wrote purely-functional implementations for each of the previously-mentioned examples and proved the relevant coherence properties so the library can be instantiated with them. The modular design of the library allows us to prove the library safe just once, and then re-use that proof to obtain safety proofs for each of the CRDTs.

Finally, we wanted to validate our claim that a client need not know whether the CRDT they are interacting with is state-based or not. We did this by using the example in Nieto et al. where they verify a client program together with an op-based positive-negative counter [20]. We were able to swap their op-based counter with our state-based positive-negative counter while leaving the proof virtually unmodified (small technical changes are required, but these could be eliminated with further Coq engineering). This shows that CRDTs can be true abstract data types, and can be specified while abstracting away implementation details.

The rest of the paper is structured as follows: Section 4 gives an overview of the CRDT resource lemmas in Nieto et al. [20], which we re-prove in the state-based context. Sections 5, 6, and 7 present STATELIB's design, specification, and safety proof, respectively. Section 8 describes the verified example CRDTs, as well as the proof of the client program that is agnostic to the CRDT class. Section 9 surveys related work and Section 10 concludes.

⁴ A vector of integers, with one entry per node in the system. The i th entry tells us how many updates have been done by replica i .

4 Background: CRDTs in Separation Logic

We give an overview of the separation logic approach to verification of op-based CRDTs in Nieto et al. [20]. Specifically, they introduce abstract separation logic resources (abstract predicates) for tracking the local and global states of a CRDT. The abstract resources are later used in the specifications of CRDT operations. Nieto et al. reason only about op-based CRDTs, but in this paper we show how to instantiate the abstract resources so that we can verify state-based CRDT implementations as well. As we will later see, this allows clients to reason about a CRDT while remaining agnostic of the CRDT’s implementation strategy.

4.1 Time, Events, and Denotations

We start by giving a few definitions that we will use throughout the paper.

Logical time allows us to order events in a distributed system using causal order. Time is axiomatized by a triple $(\text{Time}, \leq_t, <_t)$, where Time is a set of *timestamps*, \leq_t is a partial order on timestamps, and $<_t$ is the strict version of \leq_t . For example, for working with op-based CRDTs, Nieto et al. instantiate logical time by taking timestamps to be vector clocks and \leq_t to be the associated pointwise ordering.

Logical events represent operations that are executed by the CRDT, together with metadata. A logical event is a triple $(\text{op}, \text{source}, \text{time}) \in \text{Event} \triangleq \text{Op} \times \mathbb{N} \times \text{Time}$. Here Op is the type of *operations* on the CRDT (e.g. $\text{Op} \triangleq \{\text{inc}(n) \mid n \in \mathbb{N}\}$ for a g-counter), source is the id of the replica that generated the event, and time is a timestamp. For an event e we write $e.o$, $e.s$, and $e.t$ for the operation, source, and timestamp of e , respectively.

Given two event sets s and s' , we say that s is a *causally-closed subset* of s' , written $s \subseteq_{\text{cc}} s'$, if $s \subseteq s'$ and

$$\forall e e', e \in s' \Rightarrow e' \in s' \Rightarrow e.t \leq_t e'.t \Rightarrow e' \in s \Rightarrow e \in s$$

That is, if we start with two events from s' and the later one e' (according to timestamp ordering) is in s , then e (its causal dependency) must be in s as well.

Finally, we use *denotations* to specify CRDTs. A denotation is a tuple $(\text{Op}, \text{St}, s_{\text{init}} : \text{St}, \llbracket \cdot \rrbracket : \mathcal{P}(\text{Event}) \rightarrow \text{St})$. For example, as in Figure 1, for a g-counter we could have Op as previously defined, $\text{St} \triangleq \mathbb{N}$, $s_{\text{init}} \triangleq 0$, and $\llbracket s \rrbracket = \sum_{e \in s} \text{unwrap}(e.o)$ with $\text{unwrap}(\text{inc}(n)) = n$. We could have also chosen St to be the set of lists of naturals of length N , where N is the number of replicas. This latter denotation would be closer to the state-based implementation.

It is useful for denotations to be partial because we can avoid giving a meaning to ill-formed sets of events. For example, suppose we have $s = \{a, b\}$ with $a.t = b.t$ but $a \neq b$. We might know that in practice events with equal timestamps must be equal, so s can never arise during an execution. We might then choose $\llbracket s \rrbracket$ to be undefined.

4.2 Separation Logic Resources

So far we have not shown any Aneris definitions. We do so in Figure 3, which shows the types of different abstract separation logic resources (predicates) that, together with associated lemmas, can be used to reason about the state of CRDTs. Specifically, these resources appear in the pre and post-conditions of functions that operate on a CRDT. The abstract resources are designed to be generic so they can be used with multiple CRDTs. Indeed, Nieto et al. verified multiple op-based example CRDTs using these resources [20], and we have also verified multiple state-based examples. Notice that Figure 3 does not show the definition of the resources. The reader can think of Figure 3 as defining an *interface* in the

Resources (abstract predicates)

(Global invariant)	$\text{GlobInv} : iProp$
(Global state)	$\text{GlobSt} : \mathcal{P}(\text{Event}) \rightarrow iProp$
(Global snapshot)	$\text{GlobSnap} : \mathcal{P}(\text{Event}) \rightarrow iProp$
(Local state)	$\text{LocSt} : \mathbb{N} \rightarrow \mathcal{P}(\text{Event}) \Rightarrow \mathcal{P}(\text{Event}) \rightarrow iProp$
(Local snapshot)	$\text{LocSnap} : \mathbb{N} \rightarrow \mathcal{P}(\text{Event}) \Rightarrow \mathcal{P}(\text{Event}) \rightarrow iProp$

Global state laws

$$\begin{aligned} (\text{GlobStTakeSnap}) \quad & \forall E \, s, \mathcal{N}^\dagger \subseteq E \Rightarrow \text{GlobInv} \multimap \text{GlobSt}(s) \multimap \Rightarrow_E \text{GlobSt}(s) \multimap \text{GlobSnap}(s) \\ (\text{GlobSnapIncl}) \quad & \forall E \, s \, s', \mathcal{N}^\dagger \subseteq E \Rightarrow \text{GlobInv} \multimap \text{GlobSnap}(s) \multimap \text{GlobSt}(s') \multimap \Rightarrow_E s \subseteq s' \multimap \text{GlobSt}(s') \end{aligned}$$

Local state laws

$$\begin{aligned} (\text{LocSnapIncl}) \quad & \forall E \, i \, s_{\text{own}} \, s_{\text{for}} \, s'_{\text{own}} \, s'_{\text{for}}, \mathcal{N}^\dagger \subseteq E \Rightarrow \text{GlobInv} \multimap \text{LocSnap}(i, s_{\text{own}}, s_{\text{for}}) \multimap \text{LocSt}(i, s'_{\text{own}}, s'_{\text{for}}) \multimap \\ & \Rightarrow_E s_{\text{own}} \cup s_{\text{for}} \subseteq_{\text{cc}} s'_{\text{own}} \cup s'_{\text{for}} \multimap \text{LocSt}(i, s'_{\text{own}}, s'_{\text{for}}) \\ (\text{LocSnapExt}) \quad & \forall E \, i \, i' \, s_{\text{own}} \, s_{\text{for}} \, s'_{\text{own}} \, s'_{\text{for}}, \text{LocSnap}(i, s_{\text{own}}, s_{\text{for}}) \multimap \text{LocSnap}(i', s'_{\text{own}}, s'_{\text{for}}) \multimap \\ & \Rightarrow_E \forall e \, e', e \in s_{\text{own}} \cup s_{\text{for}} \Rightarrow e' \in s'_{\text{own}} \cup s'_{\text{for}} \Rightarrow e.t = e'.t \Rightarrow e = e' \\ (\text{LocSnapProv}) \quad & \forall E \, i \, s_{\text{own}} \, s_{\text{for}} \, e, \mathcal{N}^\dagger \subseteq E \Rightarrow e \in s_{\text{own}} \cup s_{\text{for}} \multimap \text{GlobInv} \multimap \text{LocSnap}(i, s_{\text{own}}, s_{\text{for}}) \multimap \\ & \Rightarrow_E \exists s_g, \text{GlobSnap}(s_g) \multimap e \in s_g \\ (\text{GlobSnapProv}) \quad & \forall E \, i \, s_{\text{own}} \, s_{\text{for}} \, s_g, \mathcal{N}^\dagger \subseteq E \Rightarrow \text{GlobInv} \multimap \text{LocSt}(i, s_{\text{own}}, s_{\text{for}}) \multimap \text{GlobSnap}(s_g) \multimap \\ & \Rightarrow_E \text{LocSt}(i, s_{\text{own}}, s_{\text{for}}) \multimap \forall e, e \in s_g \Rightarrow \text{EV_Orig}(e) = i \Rightarrow e \in s_{\text{own}} \\ (\text{Causality}) \quad & \forall E \, i \, s_{\text{own}} \, s_{\text{for}} \, s_g, \mathcal{N}^\dagger \subseteq E \Rightarrow \text{GlobInv} \multimap \text{LocSt}(i, s_{\text{own}}, s_{\text{for}}) \multimap \text{GlobSnap}(s_g) \multimap \\ & \Rightarrow_E \text{LocSt}(i, s_{\text{own}}, s_{\text{for}}) \multimap \forall e \, e', e \in s_g \Rightarrow e' \in s_{\text{own}} \cup s_{\text{for}} \Rightarrow e <_t e' \Rightarrow e \in s_{\text{own}} \cup s_{\text{for}} \end{aligned}$$

■ **Figure 3** CRDT resources and selected lemmas, from Nieto et al. [20].

software engineering sense. Nieto et al. implement this interface for op-based CRDTs, while we re-implement it for state-based CRDTs. For space reasons, we do not show the entire interface; the full interface can be found in the accompanying Coq code.

The defined resources are as follows: there is a *global invariant* GlobInv that holds throughout the CRDT’s existence. Then we define resources for tracking *global* (GlobSt) and *local* (LocSt) states. The intuition is that if $\text{GlobSt}(s_g)$ holds, then we know that s_g is *exactly* the set of events issued by any CRDT replica in the system. Similarly, ownership of $\text{LocSt}(i, s_{\text{own}}, s_{\text{for}})$ tells us that replica i has processed *exactly* the events in s_{own} and *at least* the events in s_{for} . The events in s_{own} (the “own” events) must have all originated at i , while the ones in s_{for} (the “foreign” events) must all originate outside of i . This is because a client of the CRDT knows exactly which events it has issued, but is potentially not aware of all events that have been received “in the background” since the last interaction with the CRDT.

Global and local states are *exclusive* (not shown in Figure 3), meaning that only one copy of the resource (or one per replica in the case of local state) can exist: e.g., $\text{GlobSt}(s_g) \multimap \text{GlobSt}(s'_g) \multimap \perp$.

By contrast, the interface also declares *global and local snapshots* (GlobSnap and LocSnap), which are persistent (duplicable) and give us a *lower bound* on global and local states, respectively. Snapshots are useful as a “certificate” that an event was generated by a given replica: this is the case if one can prove, e.g., that $\text{GlobSnap}(s_g)$ and $e \in s_g$ for an event e . The use of global snapshots as certificates is validated by lemma GlobSnapIncl (Figure 3). The lemma says that under the global invariant, if we own $\text{GlobSnap}(s_g)$ and $\text{GlobSt}(s'_g)$, then we must have $s_g \subseteq s'_g$. This conclusion holds under the update modality \Rightarrow_E , which

$$\begin{array}{l}
\text{INCSPEC} \\
\{\text{LocSt}(i, s_{\text{own}}, s_{\text{for}}) * \text{GlobSt}(s_{\text{g}})\} \\
\langle ip_i; \text{inc}(n) \rangle \\
\left\{ \begin{array}{l} v. \exists e s'_{\text{for}}. s'_{\text{for}} \supseteq s_{\text{for}} * e \notin s_{\text{g}} * e.o = n * e.s = i \\ \llbracket s_{\text{own}} \cup \{e\} \cup s'_{\text{for}} \rrbracket = v * \text{LocSt}(i, s_{\text{own}} \cup \{e\}, s'_{\text{for}}) * \text{GlobSt}(s_{\text{g}} \cup \{e\}) \end{array} \right\}
\end{array}$$

■ **Figure 4** Simplified spec of an increment operation, which returns the counter’s current value.

means that it holds possibly after opening (and closing) all invariants in the mask E . The premise $\mathcal{N}^\uparrow \subseteq E$ tells us that the global invariant’s namespace \mathcal{N}^\uparrow is part of E . This means that GlobInv must not be open when the lemma is called (because the proof of the lemma opens GlobInv). LocSnapIncl provides similar inclusion guarantees for local snapshots, but note that we actually get the stronger causally-closed inclusion \subseteq_{cc} , as opposed to just \subseteq .

GlobStTakeSnap allows us to take snapshots of global states. LocSnapExt says that if two events in a local snapshot have equal timestamps, then the events must be equal.

Finally, we have three lemmas that tie together local and global states. LocSnapProv says that if e is tracked locally, then there must exist some global snapshot $\text{GlobSnap}(s_{\text{g}})$ such that $e \in s_{\text{g}}$. That is, all local events are also tracked globally. GlobSnapProv says that if an event $e \in s_{\text{g}}$ has origin i and we know $\text{GlobSnap}(s_{\text{g}})$ (e is tracked globally), then e must also be in the local state for i . Finally, Causality is our definition of causality in separation logic. This take on causality was originally developed for reasoning about a causally-consistent key value store by Gondelman et al. [8], and later generalized by Nieto et al. [20] so it can apply to events in an arbitrary CRDT. The lemma is as follows: suppose we have two events e and e' such that e' was recorded locally at node i (that is, $e' \in s_{\text{own}} \cup s_{\text{for}}$ and we know $\text{LocSt}(i, s_{\text{own}}, s_{\text{for}})$). Next suppose that e happened before e' , and e is a logically tracked event (which we can show by presenting $\text{GlobSnap}(s_{\text{g}})$ with $e \in s_{\text{g}}$). Then causal delivery requires that e be observed locally at i as well: i.e., $e \in s_{\text{own}} \cup s_{\text{for}}$. Gondelman et al. [8] show how this definition of causality is strong enough to prove four *session guarantees* (monotonic reads, monotonic writes, read your writes, and writes follow reads) that programmers intuitively expect when interacting with a causally-consistent datatype.

Reasoning With Resources

To tie all the above together, Figure 4 shows how the previously-discussed resources can be used to specify the inc operation on a g-counter CRDT. We assume that inc both increments the counter and returns its current local value. The precondition for calling inc at replica i requires knowledge of both $\text{LocSt}(i, s_{\text{own}}, s_{\text{for}})$ and $\text{GlobSt}(s_{\text{g}})$. This is because every single increment must be tracked both locally at the replica where it is performed and globally. In the postcondition we get back $\text{LocSt}(i, s_{\text{own}} \cup \{e\}, s'_{\text{for}})$, where e is the event generated by the increment and $s_{\text{for}} \subseteq s'_{\text{for}}$. Notice that the “own events” grow by exactly one event, e , but the “foreign events” grow to some superset s'_{for} of s_{for} . This is because since the last time inc was called any number of new events could have been propagated from other replicas to replica i . Notice how we connect the implementation to its functional correctness specification by saying that the return value v is the denotation of the locally-observed events $s_{\text{own}} \cup \{e\} \cup s'_{\text{for}}$. Finally, observe that by using denotations we automatically obtain convergence (the safety part of SEC), because the return value is a function of a *set* of events, so two replicas that have seen the same set of events must return the same result.

Resources for State-Based CRDTs

Two difficulties arise when instantiating the resource interface for state-based CRDTs.

1. *How should we track local state?* The replica state in an implementation of a state-based CRDT is a lattice element. By contrast, the resource interface logically tracks replica states as sets of events (operations). The solution is to link the two representations via a denotation: if the logical state is $\text{LocSt}(i, s_{\text{own}}, s_{\text{for}})$, then the physical state must be $\llbracket s_{\text{own}} \cup s_{\text{for}} \rrbracket$, which in turn must be drawn from the appropriate lattice. The link is also needed when propagating a replica’s state to other replicas. In the implementation, a replica sends a message containing its entire state e , so others can merge it. Logically, we require that the sent state be paired with a local snapshot whose denotation is e .

2. *How do we track time to prove causal consistency?* For their treatment of op-based CRDTs, Nieto et al. [20] implemented a causal broadcast algorithm that ensures that every message sent by a CRDT replica is delivered in causal order. This is achieved via vector clocks in the standard way. But state-based CRDTs should not rely on causal broadcast; in fact, one of the main advantages of the state-based approach is that messages can be delivered out-of-order and re-delivered without causing issues (because of the properties of least upper bound). It is not immediately clear why the state-based design satisfies causal delivery. The first key observation is that if we start with a replica state st and look at the event set s that produced it, i.e., $\llbracket s \rrbracket = st$, then s “has no holes” with respect to causality. That is, if we take an event $e \in s$ and e' is another event that happened before e , then it must be the case that $e' \in s$ as well. This is formalized via the notion of *dependency-closure* (Section 7.1). The second observation is that when a new operation o is applied to a local state st with $\llbracket s \rrbracket = st$, it (logically) generates a new event e' with $e'.o = o$. e' ’s causal dependencies are *exactly* the events in s . This is important because it means that we can track an event’s dependencies *purely logically*, without the need for vector clocks. Using these ideas we are able to prove that the (Causality) lemma holds for state-based CRDTs (Lemma 5). To our knowledge, this is the first formal proof that state-based CRDTs are causally-consistent.

5 StateLib : a Library for Implementing State-Based CRDTs

We have structured our CRDT implementations so that common functionality is factored out into a separate library, which can then be instantiated as needed by different CRDT examples. The library, called STATELIB, is responsible for maintaining the CRDT’s internal state and inter-replica propagation. The library’s code is shown in Figure 5, together with an abridged example instantiating a *grow-only set* (g-set) CRDT, a set to which we can add elements, but from which we cannot remove them [21].

We start by describing the `init` function, which is the library’s entry point. A CRDT implementer calls `init` with the following arguments: serialization and de-serialization functions, a list of all replica addresses, the current replica id, and a `crdt` parameter that describes how the specific CRDT being instantiated should behave. The serialization functions have the expected types: `type 's serT = 's → string` and `type 's deserT = string → 's`. The `crdt` argument has type `('o, 's) crdtT`, parametrized on operations and states:

```
type 's mergeT = 's → 's → 's
type ('o, 's) mutT = int → 's → 'o → 's
type ('o, 's) crdtT = (('s * ('o, 's) mutT) * 's mergeT)
```

That is, a value of type `('o, 's) crdtT` is a triple `(init_st, mutator, merge)` containing an initial state, a mutator function, and a merge function. The mutator takes a replica id, the current state, and a new operation, and returns the state that results after applying the operation locally. The merge function takes two states and returns their least upper bound.

22:12 Modular Verification of State-Based CRDTs in Separation Logic

```

let get_state lock st () =
  acquire lock;
  let res = !st in (* LP *)
  release lock;
  res

let rec loop_forever thunk =
  thunk ();
  loop_forever thunk

let apply deser lk sh st merge :=
  loop_forever (fun () →
    let msg =
      unSOME (receiveFrom sh) in
    let st' = deser (fst msg) in
    acquire lk;
    st := merge !st st';
    release lk)

let update lk mut i st op =
  acquire lk;
  st := mut i !st op; (* LP *)
  release lk

let sendToAll sh dstl i msg =
  let j = ref 0 in
  let rec aux () =
    if !j < list_length dstl then
      if i = j then (j := !j + 1; aux ())
    else begin
      let dst =
        unSOME (list_nth dstl !j) in
      sendTo sh msg dst;
      j := !j + 1;
      aux ()
    end
  else ()
  in aux ()

let broadcast ser lk sh st dstl i =
  loop_forever (fun () →
    Unix.sleepf 2.0;
    acquire lk;
    let s = !st in
    release lk;
    let msg = ser s in
    sendToAll sh dstl i msg)

let init ser deser addr rid crdt =
  let ((init_st, mut), merge) = crdt in
  let st = ref (init_st ()) in
  let lk = newlock () in
  let sh = socket () in
  let addr = unSOME (list_nth addr rid) in
  socketBind sh addr;
  fork (apply deser lk sh st merge);
  fork (broadcast ser lk sh st addr rid);
  let get = get_state lk st in
  let upd = update lk mut rid st in
  (get, upd)

(* G-Set instantiation *)
let mutator i st op = set_add op st
let merge st1 st2 = set_union st1 st2
let init_st = set_empty
let gset_crdt = ((init_st, mutator), merge)

(* Instantiate via *)
let (get, upd) = init ... gset_crdt

```

■ **Figure 5** STATELIB implementation and a G-Set example. Linearization points are marked with an LP comment.

Back to the body of `init`, we see that it unpacks the `crdt` argument. It then allocates a local reference to hold the current state of the CRDT, a lock to control updates to the state, and a socket over which it can communicate with other replicas. The function then spawns two concurrent threads, `apply` and `broadcast`, in charge of receiving updates from other replicas and propagating local updates, respectively. Finally, `init` returns a pair of functions `get_state` and `update` allowing the library user to query and update the CRDT state.

Both `get_state` and `update` have simple implementations. The former just dereferences the local state, while the latter uses the user-provided mutator to compute the CRDT's next state. Both operations are guarded by a lock.⁵

The `apply` function, which is spawned off as a separate thread from `init`, is responsible for receiving updates from other replicas and then merging them with the local state, using the user-provided `merge` function. The call to `receiveFrom` blocks until a message is available at the given socket handle. The function `unSOME : 'a option → 'a` unwraps a value of an option type, crashing if the argument is `None`. Notice that the received message needs to be deserialized via the user-provided `deser` function. Also notice that `apply` does not terminate, but mutates the CRDT's state.

⁵ In the case of `get_state`, loads in `AnerisLang` are atomic, so the lock is not strictly needed; however, dereferences are not atomic in `OCaml` [6], which we use to run our code, so we use a lock.

The dual of `apply` is `broadcast`, which is tasked with propagating the local state to other replicas. This function also runs on a separate thread, and loops forever, working solely via side effects. The `broadcast` function retrieves a copy of the local state, serializes it, and then calls a helper function `sendToAll`. This helper takes a list of IP addresses `dst1`, the current replica id `i` and the message (state) to be sent. It then loops over the elements of `dst1` and sends the message to each of them (taking care to not send a message to itself).

Finally, Figure 5 sketches how one might implement a g-set via the library. We represent the state with a sequential set. The mutator is just set insertion, `merge` is implemented via set union, and the initial state is the empty set. We can then package these three components into a tuple `gset_crat`, and provide the latter as an argument to `init` (together with the serialization functions and replica IP addresses). From `init` we get back a pair functions for querying the current value of the set and updating it.

6 Specifying StateLib

The STATELIB library has two interfaces: an internal interface used by the CRDT *implementer*, consisting of the `init` function, and an external interface used by the CRDT *client*, consisting of `get_state` and `update`.

6.1 Internal Interface

Recall that STATELIB is initialized by the CRDT implementer through an `init` function, taking in (de-) serialization functions for the CRDT state, a list of replica IP addresses, and finally a `crat` argument describing the lattice being implemented (Figure 5). This last parameter is a triple `crat = (initSt, mut, merge)` consisting of the CRDT's initial state $\text{initSt} \in \text{LatSt}$, a *mutator* function $\text{mut} : \text{LatSt} \rightarrow \text{Event} \rightarrow \text{LatSt}$, and a *merge* function $\text{merge} : \text{LatSt} \rightarrow \text{LatSt} \rightarrow \text{LatSt}$.

The CRDT implementer first defines a poset (LatSt, \leq_L) and then proves that it is a lattice. Because our tracking of replica states is defined in terms of event sets (Figure 3) we need a way to connect said events to the physical CRDT state, which is a lattice element. Intuitively, we would like to guarantee that the CRDT's physical state is the denotation of the set of events received so far. For this to be true, we need certain coherence properties between event sets, their denotations, and lattice elements. These are shown in Figure 6 and consist of specifications for `initSt`, `mut` and `merge`.

INITSTSPEC says that the denotation of the empty set of events must be the initial state.

MUTATORSPEC says that if we start in a state $st = \llbracket s \rrbracket$ and through a mutation get to a state $st' = \text{mut}(st, op)$, then we can also arrive at st' by taking the denotation $\llbracket s \cup \{e\} \rrbracket$, where e is the event containing op . We also need to show that the mutator is monotonic: we must have $st \leq_L st'$ in the lattice order. In proving these goals we get to make additional assumptions about s and e . Specifically, we can assume that s is a set of events that is *valid with respect to coherence* (we explain validity in Section 7.1); additionally, we know that e is the maximum element with respect to timestamp ordering in the set $s \cup \{e\}$. Intuitively, this is because e is a new event being added, so it has every event in s as a causal dependency.

MERGESPEC shows coherence of merges. Here we start with two states st and st' that we want to merge to get a third state st'' . We know that $\llbracket s \rrbracket = st$ and $\llbracket s' \rrbracket = st'$, and we would like to conclude that $\llbracket s \sqcup s' \rrbracket = st''$ and also that `merge` is in fact computing the least upper bound, so $st \sqcup st' = st''$. Once again we get to assume validity of the relevant event sets, and now additionally we know an inclusion property of sections (a section is a subset of events that originates at a specific replica). The proposition `SectIncl(s, s')` tells us that if we

22:14 Modular Verification of State-Based CRDTs in Separation Logic

$$\begin{array}{l}
\text{INITSTSPEC: } \llbracket \emptyset \rrbracket = \text{initSt} \\
\left. \begin{array}{l}
\text{MUTATORSPEC} \\
\left\{ \begin{array}{l}
\llbracket s \rrbracket = st \wedge e.o = op \wedge e.s = i \wedge e \notin s \\
\wedge \text{CohVal}(s \cup \{e\}) \wedge \text{maximum}(e, s \cup \{e\})
\end{array} \right\} \\
\langle ip_i; \text{mut}(st, op) \rangle \\
\{st'. \llbracket s \cup \{e\} \rrbracket = st' \wedge st \leq_L st' \}
\end{array} \right\} \quad \left. \begin{array}{l}
\text{MERGESPEC} \\
\left\{ \begin{array}{l}
\llbracket s_1 \rrbracket = st_1 \wedge \llbracket s_2 \rrbracket = st_2 \wedge \text{SectIncl}(s_1, s_2) \\
\wedge \text{CohVal}(s_1) \wedge \text{CohVal}(s_2) \wedge \text{CohVal}(s_1 \cup s_2)
\end{array} \right\} \\
\langle ip_i; \text{merge}(st_1, st_2) \rangle \\
\{st'. st_1 \sqcup st_2 = st' \wedge \llbracket s_1 \cup s_2 \rrbracket = st' \}
\end{array} \right\} \\
\text{sect}(s, i) \triangleq \{e \in s \mid e.s = i\} \\
\text{SectIncl}(s, s') \triangleq \forall i. \text{sect}(s, i) \subseteq \text{sect}(s', i) \vee \text{sect}(s', i) \subseteq \text{sect}(s, i) \\
\text{CohVal}(s) \text{ is a version of "local state validity" (Section 7.1) that does not imply } \text{depcoed}(s). \\
\text{CRDTSPEC}(\text{initSt}, \text{mut}, \text{merge}) \triangleq \text{INITSTSPEC}(\text{initSt}) * \text{MUTATORSPEC}(\text{mut}) * \text{MERGESPEC}(\text{merge})
\end{array}$$

$$\begin{array}{l}
\text{INITSPEC} \\
\{ \dots * \text{CRDTSPEC}(\text{crdt}) \} \\
\langle ip_i; \text{init}(\text{addrs}, \text{repld}, \text{crdt}) \rangle \\
\{ (\text{get_state}, \text{update}). \text{LocSt}(i, \emptyset, \emptyset) * \text{GETSTATESPEC}(\text{get_state}) * \text{UPDATESPEC}(\text{update}) \}
\end{array}$$

■ **Figure 6** Internal specifications. GETSTATESPEC and UPDATESPEC are defined in Figure 7.

look at any particular section, say section i , then either $\text{sect}(s, i)$ is a subset of $\text{sect}(s', i)$, or the other way around. Intuitively, this is because sections are always “complete”: if a replica has received event $(6, 5)$, it must have also received all events in the range $(6, 1), \dots, (6, 4)$.

We package the three specifications in the assertion $\text{CRDTSPEC}(\text{crdt})$, which asserts that each of the components of the crdt tuple satisfies the corresponding spec above. Finally we have specification for the init function. In the precondition of INITSPEC , we assert that the crdt argument satisfies CRDTSPEC . In the postcondition, we learn that init returns a pair of functions get_state and update that satisfy the same-named specifications (described in the next section). We also gain ownership of the resource $\text{LocSt}(i, \emptyset, \emptyset)$, indicating that the local replica has yet to receive any events (because it has just been initialized).

6.2 External Interface

STATELIB’s external interface consists of two functions: get_state , which takes no arguments and returns the local state of the CRDT, and update , which takes an operation, updates the local state, and returns a Unit . Figure 7 shows specifications for both functions; these specifications are identical to the ones for the same-named functions in Nieto et al.’s library for op-based CRDTs [20]. This is by design: by proving that our library meets the same specification as the equivalent library for op-based CRDTs we then make it possible for client programs to use (and reason about) a replicated data type without knowledge of whether the data type is op-based or state-based. That is, we hide implementation details to turn CRDTs into true abstract data types.

Looking at Figure 7, the reader will notice that the specifications use angle brackets instead of braces: i.e. we write $\langle P \rangle e \langle Q \rangle^N$, instead of the usual Hoare triple $\{P\}e\{Q\}$. The former is a *logically-atomic* triple [10]. The motivation for these triples is that Aneris invariants can only be opened around *atomic* steps (otherwise concurrent threads might observe invariant violations); this means we cannot use a specification $\{P\}e\{Q\}$ if we need to open an invariant to prove P , provided e is not atomic, as is the case for both get_state and update . In particular, the precondition of update requires the global state $\text{GlobSt}(s_g)$, which a client is likely to keep in an invariant because it is shared by all (concurrent) replicas.

$$\begin{array}{l}
\text{GETSTATESPEC} \\
\langle \text{LocSt}(i, s_{\text{own}}, s_{\text{for}}) \rangle \\
\langle ip_i; \text{get_state}() \rangle \\
\left\langle \begin{array}{l} v. \exists s'_{\text{for}} w. s'_{\text{for}} \supseteq s_{\text{for}} * \text{StCoh}(w, v) * \\ \text{LocSt}(i, s_{\text{own}}, s'_{\text{for}}) * \llbracket s_{\text{own}} \cup s'_{\text{for}} \rrbracket = w \end{array} \right\rangle \mathcal{N}
\end{array}
\quad
\begin{array}{l}
\text{UPDATESPEC} \\
\langle \text{LocSt}(i, s_{\text{own}}, s_{\text{for}}) * \text{GlobSt}(s_{\text{g}}) \rangle \\
\langle ip_i; \text{update}(v) \rangle \\
\left\langle \begin{array}{l} (). \exists e s'_{\text{for}}. s'_{\text{for}} \supseteq s_{\text{for}} * e \notin s_{\text{own}} * e \notin s_{\text{g}} * e.o = v * \\ e.s = i * e \in \text{maximals}(s_{\text{g}} \cup \{e\}) * \\ \text{maximum}(e, s_{\text{own}} \cup s'_{\text{for}} \cup \{e\}) * \\ \text{LocSt}(i, s_{\text{own}} \cup \{e\}, s'_{\text{for}}) * \text{GlobSt}(s_{\text{g}} \cup \{e\}) \end{array} \right\rangle \mathcal{N}
\end{array}$$

■ **Figure 7** External specifications. \mathcal{N} is any invariant namespace that includes `GlobInv`'s name. Adapted from Nieto et al. [20].

Logically-atomic triples solve this problem by allowing us to open invariants when proving the pre-condition. Their informal semantics are as follows: if we know $\langle P \rangle e \langle Q \rangle^{\mathcal{N}}$, then we know that e executes without crashing (although it might not terminate) provided that P holds. When proving P we are allowed to use any invariants that are not in namespace \mathcal{N} .⁶ We also need to show that if Q holds we can close any invariants that were open when proving P . Another point of view is that P and Q hold around a *linearization point* in e ; the linearization points for `get_state` and `update` are marked with an LP comment in Figure 5.

The specification of `get_state` can be read as follows. Before calling `get_state` we should know the local state at replica i : $\text{LocSt}(i, s_{\text{own}}, s_{\text{for}})$; afterwards, the function returns a physical state v which is *coherent* with a logical state w ,⁷ the local state is now $\text{LocSt}(i, s_{\text{own}}, s'_{\text{for}})$ where $s'_{\text{for}} \supseteq s_{\text{for}}$ (reflecting the fact the set of local events is unchanged, but additional remote events might have been received), and the returned value w is the denotation of $s_{\text{own}} \cup s'_{\text{for}}$. Notice that the fact that the returned value is a function of the set of received events automatically gives us the safety part of eventual consistency (convergence): if two replicas have received the same set of events, then they are in the same state.

The specification of `update` says that we need to know both the local state $\text{LocSt}(i, s_{\text{own}}, s_{\text{for}})$ at replica i and the global state $\text{GlobSt}(s_{\text{g}})$. This is because every event needs to be tracked both locally and globally. The function does not return any meaningful value, but we do get (logical) knowledge that the set of events has expanded: locally we now know $\text{LocSt}(i, s_{\text{own}} \cup \{e\}, s'_{\text{for}})$ with $s_{\text{for}} \subseteq s'_{\text{for}}$, and globally we have $\text{GlobSt}(s_{\text{g}} \cup \{e\})$. The new event e ($e \notin s_{\text{g}}$) has the value we are updating by as its payload ($e.o = v$) and it originates at replica i . Finally, we know that this new event is more recent than any other locally-received event ($\text{maximum}(e, s_{\text{own}} \cup s'_{\text{for}} \cup \{e\})$), and we also know that no other event (even globally) has the new event as its dependency: $e \in \text{maximals}(s_{\text{g}} \cup \{e\})$.

7 Verifying StateLib

To prove that STATELIB meets its external specification we follow a proof methodology inspired by previous Aneris developments [8, 20]:

1. We first model the CRDT as a state-transition system (STS), where the STS states are tuples detailing the state of the CRDT both globally and at each replica (Section 7.1). Transitions correspond to mutations and merges. Crucially, we show that transitions preserve *state validity*, a safety invariant from which we can derive properties of interest (e.g., causality). This is done in the meta-logic (i.e., Coq) and outside of separation logic.

⁶ This is to prevent a user of STATELIB from opening the global invariant, which is needed by STATELIB. Invariants cannot be reopened, to preserve soundness of the logic.

⁷ The *state coherence* predicate $\text{StCoh}(w, v)$ links the physical and logical states. This is useful when the physical state has a more involved representation due to limitations of AnerisLang: for example, v might be a pair of pairs while w is a 3-tuple, because AnerisLang only supports pairs.

2. We then embed the STS model inside Aneris via a combination of invariants and ghost state (defined via PCMs, see Section 7.3). We use state validity and the properties of the relevant PCMs to show that the resource interface from Section 4 holds.
3. Finally, using the separation logic resources defined in the previous step, we prove that STATELIB’s implementation meets its external specification (Section 7.4).

The rest of Section 7 is technical in nature, so the reader interested in an overview of our work can skip to Section 8. We highlight Lemmas 4 and 5, which, to our knowledge, are the first formal proofs that state-based CRDTs are causally-consistent.

7.1 State-Transition System Model

We model the execution of a CRDT via an STS that keeps track of the per-replica state, as well as the global state. We then show a number of safety invariants that hold for any execution of the STS. In later sections we show how the `AnerisLang` implementation simulates the STS, therefore inheriting its safety properties. We use the simulation to prove the resource laws in Figure 3.

The reader might wonder why we develop this STS model when state-based CRDTs already have a well-understood model that is lattice-based. We do this because our goal is to prove that STATELIB satisfies general functional correctness specifications that apply to both state-based CRDTs and op-based CRDTs. To this end we write our high-level specifications in terms of denotations, which talk about sets of events instead of lattice elements. This is why we need the STS model below. At an operational level, the STS model is needed to define the ghost state in Section 7.3 and as such is not directly exposed to the user.

We start by defining a purely logical notion of time that allows us to reason about causality in the absence of vector clocks. *Logical time* for state-based CRDTs is a triple $\text{LogTime}_{\text{st}} \triangleq (\mathcal{P}(\text{EvtId}), \subseteq, \prec)$, where $\text{EvtId} \triangleq \mathbb{N} \times \mathbb{N}$. Here EvtId is the set of *event ids*, which are pairs (r, n) of a *replica id* and a *sequence number*, respectively. We show that $\text{LogTime}_{\text{st}}$ satisfies the requirements on logical time from Section 4.1.

Given a set $d \in \mathcal{P}(\text{EvtId})$ of event ids we can extract the subset that originates at a given replica id via a *section*: $\text{sect}(d, i) \triangleq \{(i, n) \mid (i, n) \in d\}$. We can also compare event ids, but only if they are in the same section: $(s, n) \leq (s, n') \triangleq n \leq n'$.

We define logical events as triples $(\text{op}, \text{src}, \text{time}) \in \text{Op} \times \mathbb{N} \times \mathcal{P}(\text{EvtId})$. We tag each event e with the set of event ids of its *causal dependencies* $e.t$ and are then able to sort events according to causal order. For example, if $e.t = \{(1, 1)\}$ and $e'.t = \{(1, 1), (2, 1)\}$, then $e.t <_t e'.t$. Let $s \in \mathcal{P}(\text{Event})$. We lift causal dependencies to sets of events: $\text{deps}(s) \triangleq \bigcup_{e \in s} e.t$.

The *id* of an event e can be computed by counting the number of dependencies that originate at e ’s origin: $\text{id}(e) = (e.s, |\text{sect}(e.t, e.s)|)$. This way of computing event ids makes sense only if we assume that sequence numbers (a) start at 1 and (b) there are no “holes” in the ids stored in $e.t$. In our proof we maintain an invariant that implies these two properties.

To ensure causal consistency, we care about event sets that are closed with respect to causal dependencies. Let s be a set of events, then s is *dep-closed*, written $\text{depclosed}(s)$, if $\forall e \in s, \text{id} \in e.t, \exists e' \in s, \text{id}(e') = \text{id}$. For example, if $e \in s$ and $(1, 2) \in e.t$, then we must have $e' \in s$ with $\text{id}(e') = (1, 2)$. Dep-closure is preserved by set union, which is key because it will allow us to link logical and physical states when we take least upper bounds of the latter.

We now define *local states*, which track the state of the CRDT at a given replica. Unlike in the implementation the replica state will not be a lattice element, but a set of events: $\text{Lst} = \mathcal{P}(\text{Event})$. We lift sections to local states: if $s \in \text{Lst}$ then $\text{sect}(s, i) \triangleq \{e \mid e.s = i\}$.

Recall that $\text{numRep} \in \mathbb{N}$ denotes the number of replicas. We define *global states* $\text{Gst} \triangleq \mathcal{P}(\text{Event}) \times \text{Lst}^{\text{numRep}}$. The intuition for a global state $(s_g, \vec{s}_l) \in \text{Gst}$ is that the first component s_g gives us a *global view* of the system (we will ensure that s_g equals the union of all $s_{l,i}$). The second component \vec{s}_l is a vector of length numRep containing the local state at each replica.

► **Definition 1** (State-transition system model). *The state-transition system model is $\mathcal{S} = (\text{Gst}, \text{init}_{\mathcal{S}}, \rightarrow_{\mathcal{S}})$. STS states are elements of Gst and $\text{init}_{\mathcal{S}} \triangleq (\emptyset, \vec{\emptyset})$ is the initial state. The transition relation $\rightarrow_{\mathcal{S}} \subseteq \text{Gst} \times \text{Gst}$ is defined by the following two inference rules:⁸*

$$\frac{s_{l,i} = s \quad d = \text{deps}(s) \quad n = |\text{sect}(d, i)| + 1 \quad t = \{(i, n)\} \cup d \quad e = (\text{op}, i, t)}{(s_g, \vec{s}_l) \rightarrow_{\mathcal{S}} (s_g \cup \{e\}, \vec{s}_l[i \mapsto s \cup \{e\}])} \text{TUpdate}$$

$$\frac{s \subseteq s_{l,j} \quad \text{depclosed}(s)}{(s_g, \vec{l}) \rightarrow_{\mathcal{S}} (s_g, \vec{l}[i \mapsto s_{l,i} \cup s])} \text{TMerge}$$

The TUpdate rule models the execution of a new operation at a particular replica. The premises say that the local state at replica i is s and d is the set of dependencies of all events in s . Then we compute the sequence number for the new event: since it originates in i this needs to be exactly one larger than the number of dependencies in d that come from i , hence $n = |\text{sect}(d, i)| + 1$. Then we build a timestamp for the new event: every (old) event in s should be a causal dependency of the new event, plus the new event's id is also a dependency, so $t = \{(i, n)\} \cup d$. Finally we build the new event $e = (\text{op}, i, t)$. Given all the above, we can take a step in the STS from a state (s_g, \vec{s}_l) to a state that includes the new event e . Because e is new, it should be added both to the global state and the local state for i . The notation $\vec{s}_l[i \mapsto s']$ stands for the vector that is like \vec{s}_l except that the i 'th entry is now s' .

The TMerge rule models merge operations where a replica updates its state by receiving and merging a (potentially old) state that was sent by another replica. In the rule, we start with some subset s of the local state at replica j . It is crucial that said subset be dep-closed so that we can preserve causality. In the rule's conclusion, we merge s with the state at replica i . That is, we can think of this rule as saying that replica j transmitted its state to replica i , which subsequently merged it. Finally, notice that the fact that s is a subset of $s_{l,j}$ and not exactly $s_{l,j}$ allows us to model the delay imposed by the network on message transmission – the fact that s is a *dep-closed* subset of $s_{l,j}$ means that s is a version of the state of the j^{th} replica from the past.

7.2 Safety Invariants

The goal of the STS model is to allow us to show a number of safety invariants about the execution of the system. We do this through the notions of *local* and *global state validity*. Let $s \in \text{Lst}$. Then s is a *valid local state*, written $\text{LocStValid}(s)$, if all the following hold:

⁸ As usual $\rightarrow_{\mathcal{S}}^*$ denotes the reflexive transitive closure of $\rightarrow_{\mathcal{S}}$.

(DepClosed)	$\text{depclosed}(s)$
(SameOrigComp)	$\forall i, \forall e' \in \text{sect}(s, i), e.t <_t e'.t \vee e.t = e'.t \vee e'.t <_t e.t$
(ExtId)	$\forall e' \in s, \text{id}(e) = \text{id}(e') \Rightarrow e = e'$
(ExtTime)	$\forall e' \in s, e.t = e'.t \Rightarrow e = e'$
(OrigRange)	$\forall e \in s, e.s < \text{numRep}$
(SeqIdComplete)	$\forall e \in s, n \in \mathbb{N}, 0 < (e.s, n) \leq \text{id}(e) \Rightarrow (e.s, n) \in e.t$
(SeqIdNon0)	$\forall e \in s, r, n \in \mathbb{N}, \text{id}(e) = (r, n) \Rightarrow 0 < n$
(EvIdMon)	$\forall e' \in s, e.s = e'.s \Rightarrow e.t \leq_t e'.t \Rightarrow \text{id}(e) \leq \text{id}(e')$
(EvIdIncl)	$\forall e \in s, \text{id}(e) \in e.t$
(EvIdTime)	$\forall e' \in s, \text{id}(e) \in e'.t \Rightarrow e.t \leq_t e'.t$

The different requirements on valid local states are as follows. (DepClosed) requires that a valid state s be also dep-closed. (SameOrigComp) says that events with the same origin can be totally ordered by timestamp ordering. (ExtId) and (ExtTime) say that events with equal id, resp. timestamp, must be equal. (OrigRange) ensures that replica ids are in the expected range. (SeqIdComplete) says that if $e \in s$ and e 's id is e.g. $(4, 10)$, then all timestamps in the range $(4, 1) \dots (4, 9)$ must also be in e 's dependencies. (SeqIdNon0) says that all event ids have a sequence number that starts at 1. (EvIdMon) requires that timestamps ordering and event id ordering agree. (EvIdIncl) says that an event id must be included in the event's dependencies. Finally, (EvIdTime) requires that if e 's id is in the dependencies of e' then e must have in fact happened before e' according to timestamp ordering. The definition of local state validity has been simplified with respect to prior developments [8, 20]. This is because these works use vector clocks as their notion of logical and physical time, whereas we only track time logically via sets of dependencies.

From local state validity we obtain a number of derived lemmas. For example, we can relate dep-closed and causally-closed subsets:

► **Lemma 2.** *Let $s \subseteq s_g$ where $\text{depclosed}(s)$ and $\text{LocStValid}(s_g)$. Then $s \subseteq_{\text{cc}} s_g$.*

Now we define validity of global states. Let $q = (s_g, \vec{s}_l) \in \text{Gst}$. Then q is a *valid global state*, written $\text{GlobStValid}(q)$, if all the following hold:

(InclLocal)	$s_g = \bigcup_{1 \leq i \leq \text{numRep}} s_{l, i}$
(InclOrig)	$\forall e \in s_g, e \in s_{l, e.s}$
(GlobValid)	$\text{LocStValid}(s_g)$
(LocValid)	$\forall 1 \leq i \leq \text{numRep}, \text{LocStValid}(s_{l, i})$

Given a global state (s_g, \vec{s}_l) , global state validity amounts to requiring that s_g be the union of the $s_{l, i}$ (InclLocal), that events be present in the local state from which they putatively originate (InclOrig), that the global state s_g itself be valid if treated as a local state (GlobValid) and that each local state be valid (LocValid). The definition of global state validity is essentially unchanged from prior work [8, 20].

The definitions of local and global state validity are motivated by two desiderata: they must be *invariants*, i.e. hold for all reachable states (including the initial state); and they must imply the CRDT resource lemmas from Figure 3 “at the model level”.

► **Theorem 3 (Validity invariant).** *Let $q \in \text{Gst}$ such that $\text{init}_S \rightarrow_S^* q$. Then $\text{GlobStValid}(q)$.*

We use validity to show model-level counterparts of the lemmas in Figure 3. Intuitively, ownership of $\text{GlobSt}(s_g)$ tells us that the global state is (s_g, \vec{s}_l) , whereas if we know $\text{GlobSnap}(s'_g)$ all we can say is that $s'_g \subseteq s_g$. Similarly, ownership of $\text{LocSt}(i, s_{\text{own}}, s_{\text{for}})$

corresponds to knowing that the local state $s_{l,i}$ (which we can assume to be valid) equals $s_{\text{own}} \dot{\cup} s'_{\text{for}}$ (disjoint union), with $s_{\text{for}} \subseteq s'_{\text{for}}$. The local snapshot $\text{LocSnap}(i, s_{\text{own}}, s_{\text{for}})$ give us $s_{\text{own}} \dot{\cup} s_{\text{for}} \subseteq s_{l,i}$. With this analogy in mind, here is the model-level version of causality.

► **Lemma 4** (Model-level causality). *Let $q = (s_g, \vec{s}_i)$ and $\text{GlobStValid}(q)$. Also let $e \in s \subseteq_{\text{cc}} s_{l,i}$ and $e' \in s_g$, with $e'.t <_t e.t$. Then $e' \in s$.*

Proof. From $\text{GlobStValid}(q)$ we can conclude $\text{LocStValid}(s_{l,i})$, which in turn gives us $\text{depclosed}(s_{l,i})$. Since $s_{l,i} \subseteq s_g$, by Lemma 2 we get $s_{l,i} \subseteq_{\text{cc}} s_g$. Since we assumed $s \subseteq_{\text{cc}} s_{l,i}$, we can use transitivity of \subseteq_{cc} to conclude $s \subseteq_{\text{cc}} s_g$. This implies the conclusion. ◀

7.3 Separation Logic Encoding

The next step in the proof is to encode the validity invariant using separation logic. We do this using a combination of Iris invariants and resources [9]. Recall that Iris invariants are propositions that hold throughout the execution of the operational semantics and resources are elements of partial commutative monoids (PCMs).

Here we use resources whose ownership reveals what state the system is partially in (e.g. the set of events received by a specific replica). In particular, we use three main PCM constructions, which we review below:⁹

1. The *authoritative* PCM $\text{AUTH}(\mathbb{M})$, where \mathbb{M} is itself a PCM. Given a PCM \mathbb{X} , we can define the *extension order* on elements of the carrier as follows: $x \leq_{\mathbb{X}} y \triangleq \exists z, x \cdot z = y$. The authoritative construction gives us two kinds of resources: a *full part* $\bullet_{\mathbb{M}} g$ and one or more *fragmental parts* $\circ_{\mathbb{M}} s$, where $g, s \in \mathbb{M}$. Ownership of the full part is exclusive, while ownership of a fragment $\circ_{\mathbb{M}} s$ is exclusive or persistent depending on whether ownership of s is exclusive or persistent in \mathbb{M} . The fragmental parts are guaranteed to be smaller than the full part according to extension order, so that if we own $\{\bullet_{\mathbb{M}} g\}^\gamma * \{\circ_{\mathbb{M}} s\}^\gamma$ we can conclude $s \leq_{\mathbb{M}} g$. We also have that $\circ_{\mathbb{M}} g \cdot \circ_{\mathbb{M}} s = \circ_{\mathbb{M}} (g \cdot s)$.
2. The *fractional* PCM $\text{FRAC}(X)$, where X is a carrier set. Elements of this monoid are of the form s^p , where $s \in X$ and $p \in \mathbb{Q}_{(0,1]}$. This PCM allows us to split and re-combine fractions of a resource: $s^{p+q} = s^p \cdot s^q$. We also know that if we own multiple fractions then they must add to less than 1. This is useful to e.g. make a proposition exclusive (non-duplicable) by defining it as a fraction greater than $\frac{1}{2}$ as no two copies of such a resource can be owned separately. We also know that all fractions agree on the underlying element: $\{s^p\}^\gamma * \{r^q\}^\gamma$ implies $s = r$.
3. The *monotone* PCM $\text{MONO}(R)$, where $R \subseteq X \times X$ is a pre-order on a carrier set X [22]. This PCM allows us to lift R to the extension order of $\text{MONO}(R)$: any $x \in X$ can be injected into $\text{MONO}(R)$ via a principal_R function such that $xRy \iff \text{principal}_R(x) \leq_{\text{MONO}(R)} \text{principal}_R(y)$. Combining this with the authoritative PCM gives us a monoid $\text{AUTH}(\text{MONO}(R))$ where if we know $\{\bullet_{\mathbb{M}} \text{principal}_R(g)\}^\gamma * \{\circ_{\mathbb{M}} \text{principal}_R(s)\}^\gamma$ we can conclude sRg . We instantiate this construction with $R = \subseteq_{\text{cc}}$.

We use the defined invariants and resources to prove the interface described in Figure 3. In this section, we sketch out proofs for some of the interface lemmas.

⁹ We use a few additional PCMs in the Coq formalization but elide those additional structures here for the sake of brevity.

The global invariant uses the predicate **GI** below. The predicate states that there exists a (model-level) global state h which is valid. Furthermore, it asserts ownership of global and local resources defined by the predicates $\mathbf{GR}(s_g)$ and $\mathbf{LR}(i, s_{l,i})$, respectively.

$$\mathbf{GI} \triangleq \exists h \in \overline{\mathbf{Gst}}. h = (s_g, \vec{s}_i) * \mathbf{GlobStValid}(h) * \mathbf{GR}(s_g) * \bigstar_{i=1}^{\text{numRep}} \mathbf{LR}(i, s_{l,i})$$

Given the above definition and an invariant name ι , we can instantiate the $\mathbf{GlobInv}$ predicate from Figure 3 by allocating an Aneris invariant stating that **GI** holds after every execution step: $\mathbf{GlobInv} \triangleq \overline{\mathbf{GI}}^\iota$.

The *global resource* predicate $\mathbf{GR}(s_g)$ asserts ownership of two pieces of ghost state, both of which precisely track the value of s_g : $\mathbf{GR}(s_g) \triangleq \overline{\left[\frac{1}{3} \right]}^{\gamma_{\text{gst}}} * \overline{\left[\bullet_{\mathbb{S}} s_g \right]}^{\gamma_{\text{gsnap}}}$.

The ghost state $\frac{1}{3}$ is drawn from the $\mathbf{FRAC}(\mathbf{Gst})$ PCM. Its purpose is to track the global set of events. The remaining $\frac{2}{3}$ fraction is kept *outside* of the invariant as the user-facing resource \mathbf{GlobSt} from Figure 3: $\mathbf{GlobSt}(s_g) \triangleq \overline{\left[\frac{2}{3} \right]}^{\gamma_{\text{gst}}}$. Because the fraction in \mathbf{GlobSt} is greater than a half, we can prove that $\mathbf{GlobSt}(s_g)$ is exclusive ($\mathbf{GlobStExcl}$, Figure 3).

The second part of $\mathbf{GR}(s_g)$ asserts ownership of $\bullet_{\mathbb{S}} s_g$. Here, \mathbb{S} is the PCM of finite sets of events, with set union as composition. This means that $p \leq_{\mathbb{S}} q$ iff $p \subseteq q$. Consequently, $\overline{\left[\bullet_{\mathbb{S}} s_g \right]}^{\gamma_{\text{gsnap}}} * \overline{\left[\bullet_{\mathbb{S}} s'_g \right]}^{\gamma_{\text{gsnap}}}$ implies $s'_g \subseteq s_g$. We keep the full part in the invariant and use the fragmental part to define global snapshots: $\mathbf{GlobSnap}(s) \triangleq \overline{\left[\bullet_{\mathbb{S}} s \right]}^{\gamma_{\text{gsnap}}}$. Note that these fragmental parts are persistent (duplicable) as the set union operation is idempotent.

The next step is to define the local resources predicate $\mathbf{LR}(i, s)$ which tracks in the invariant the local resources for replica i :

$$\begin{aligned} \mathbf{LR}(i, s) \triangleq & \exists s_{\text{own}} s_{\text{for}} s_{\text{sub}}, s = s_{\text{own}} \cup s_{\text{for}} * s_{\text{own}} \cup s_{\text{sub}} \subseteq_{\text{cc}} s_{\text{own}} \cup s_{\text{for}} \\ & * \mathbf{LocEv}(i, s_{\text{own}}) * \mathbf{ForEv}(i, s_{\text{for}}) * \mathbf{ForEv}(i, s_{\text{sub}}) * \overline{\left[\frac{1}{3} \right]}^{\gamma_{\text{own}_i}} * \overline{\left[\frac{1}{2} \right]}^{\gamma_{\text{for}_i}} * \overline{\left[\frac{1}{3} \right]}^{\gamma_{\text{sub}_i}} \\ & * \overline{\left[\bullet_{\mathbb{M}} \text{principal}_{\subseteq_{\text{cc}}} (s_{\text{own}} \cup s_{\text{for}}) \right]}^{\gamma_{\text{ccfor}_i}} * \overline{\left[\bullet_{\mathbb{M}} \text{principal}_{\subseteq_{\text{cc}}} (s_{\text{own}} \cup s_{\text{sub}}) \right]}^{\gamma_{\text{ccsub}_i}} \end{aligned}$$

The predicate $\mathbf{LR}(i, s)$ says that s can be broken up into two (disjoint) sets s_{own} and s_{for} . The sets are disjoint because every event in s_{own} originates at replica i , whereas all events in s_{for} originate outside of replica i . This is expressed by the predicates $\mathbf{LocEv}(i, p) = \forall e \in p. e.s = i$ and $\mathbf{ForEv}(i, p) = \forall e \in p. e.s \neq i$, respectively. Additionally, there is a third set s_{sub} which is a subset of s_{for} (this is implied by $s_{\text{own}} \cup s_{\text{sub}} \subseteq_{\text{cc}} s_{\text{own}} \cup s_{\text{for}}$). The intuition for s_{own} and s_{for} is that they precisely track the set of events that have been delivered at replica i and originate at i or outside of i , respectively. However, the user at replica i is not aware of all those events: specifically, while the user is aware of (the effects of) all its local events, it might not have observed all events that originate outside of i . The set s_{sub} precisely tracks the set of remote events, $\mathbf{ForEv}(i, s_{\text{sub}})$, that replica i has observed and is aware of. The tracking of all these event sets is precise because of the ownership of the fractions $\overline{\left[\frac{1}{3} \right]}^{\gamma_{\text{own}_i}} * \overline{\left[\frac{1}{2} \right]}^{\gamma_{\text{for}_i}} * \overline{\left[\frac{1}{3} \right]}^{\gamma_{\text{sub}_i}}$. Additionally, the invariant has “read-only” access to the three pieces of ghost state because it does not possess the full fractions. Before pressing on with the definition of $\mathbf{LR}(i, s)$, let us look at the definition of local state and snapshot from Figure 3:

$$\begin{aligned} \mathbf{LocSt}(i, s_{\text{own}}, s_{\text{sub}}) \triangleq & \overline{\left[\frac{1}{3} \right]}^{\gamma_{\text{own}_i}} * \overline{\left[\frac{2}{3} \right]}^{\gamma_{\text{sub}_i}} * \mathbf{LocSnap}(i, s_{\text{own}}, s_{\text{sub}}) \\ \mathbf{LocSnap}(i, s_{\text{own}}, s_{\text{sub}}) \triangleq & \mathbf{LocEv}(i, s_{\text{own}}) * \mathbf{ForEv}(i, s_{\text{sub}}) * \overline{\left[\bullet_{\mathbb{M}} \text{principal}_{\subseteq_{\text{cc}}} (s_{\text{own}} \cup s_{\text{sub}}) \right]}^{\gamma_{\text{ccsub}_i}} \end{aligned}$$

Notice that ownership of the local state resource gives us knowledge of some but not all of the missing fractions for s_{own} and s_{sub} : $\frac{1}{3}\gamma_{\text{own}_i} * \frac{2}{3}\gamma_{\text{sub}_i}$. This corresponds to our intuition that s_{sub} tracks the set of events the user is aware of. Notice that local state does not contain a fraction of s_{for} , because otherwise the library could not accept remote updates in a background thread.

We next explain the use of the PCM $\mathbb{M} \triangleq \text{MONO}(\subseteq_{\text{cc}})$, where $\subseteq_{\text{cc}}: \text{Event} \times \text{Event}$. We use two instances of this PCM, each under a different family of ghost names: γ_{ccsub_i} and γ_{ccfor_i} . The local state contains just γ_{ccsub_i} and the invariant contains both γ_{ccsub_i} and γ_{ccfor_i} . The intuition for holding the fragmental part $\circ_{\mathbb{M}} \text{principal}_{\subseteq_{\text{cc}}}(s_{\text{own}} \cup s_{\text{sub}})$ is to allow us to prove inclusion of local snapshots (`LocSnapIncl`, Figure 3).

We can now sketch the proof of the causality lemma. This proof is representative of our methodology: use the properties of the different PCMs to identify parts of a (valid) global state we are currently in, and then rely on a model-level lemma to get the result we want.

► **Lemma 5** (Causality, Figure 3). $\text{GlobInv} \multimap \text{LocSt}(i, o, s) \multimap \text{GlobSnap}(h) \multimap \text{LocSt}(i, o, s) * \forall e e', e \in h \Rightarrow e' \in o \cup s \Rightarrow e <_t e' \Rightarrow e \in o \cup s$.

Proof. We open the global invariant and learn that the current global state (s_g, \bar{s}_l) is valid. We also obtain global resources $\frac{1}{3}\gamma_{\text{gst}} * \bullet_{\text{S}} s_g^{\gamma_{\text{gsnap}}}$ and local resources $\frac{1}{3}\gamma_{\text{own}_i} * \frac{2}{3}\gamma_{\text{sub}_i}$, where $s_{\text{own}} \cup s_{\text{sub}} \subseteq_{\text{cc}} s_{\text{own}} \cup s_{\text{for}} = s_{l,i}$. From `LocSt` (i, o, s) we get $\frac{1}{3}\gamma_{\text{own}_i} * \frac{2}{3}\gamma_{\text{sub}_i}$, which tells us that $o = s_{\text{own}}$ and $s = s_{\text{sub}}$. From `GlobSnap` (h) we get $\frac{1}{3}\gamma_{\text{own}_i} * \frac{2}{3}\gamma_{\text{sub}_i}$, which tells us that $h \subseteq s_g$. This means we have $e \in s_g$ and e' is in a causally-closed subset of $s_{l,i}$, so we can finish by applying Lemma 4. ◀

There are additional two resources in the definition of $\mathbf{LR}(i, s)$: $\frac{1}{2}\gamma_{\text{for}_i}$ and $\bullet_{\mathbb{M}} \text{principal}_{\subseteq_{\text{cc}}}(s_{\text{own}} \cup s_{\text{for}})^{\gamma_{\text{ccfor}_i}}$. These are used in the definition of the lock invariant and socket protocol, respectively. We explain them in the next section.

In addition to proving the resource lemmas from Figure 3, we also need to show that the PCMs we have chosen can make frame-preserving updates that are compatible with the two transitions (`TUpdate` and `TMerge`) from Definition 1. We refer to reader to our Coq formalization for details.

7.4 Safety Proof

STATELIB's `init` function allocates a reference with the CRDT's initial state and spawns two concurrent threads: `apply` receives states from other replicas and merges them with the current state, and `broadcast` regularly sends the current state to all other replicas. Access to the (shared) local state is coordinated via a spinlock. The associated *lock invariant* [1], defined by the predicate $\mathbf{LI}(i, \ell)$ below, is the key ingredient of the library's safety proof (ℓ is the memory location holding the CRDT's state). When a thread acquires the lock, it gets to assume $\mathbf{LI}(i, \ell)$; unlike a regular invariant, which needs to be restored after a single atomic step, a lock invariant need not be restored until the thread releases the lock.

$$\begin{aligned} \mathbf{LI}(i, \ell) \triangleq \exists st s_{\text{own}} s_{\text{for}}. \text{LocEv}(i, s_{\text{own}}) * \text{ForEv}(i, s_{\text{for}}) * \ell \mapsto_{\text{ip}} st \\ * \llbracket s_{\text{own}} \cup s_{\text{for}} \rrbracket = st * \frac{1}{3}\gamma_{\text{own}_i} * \frac{1}{2}\gamma_{\text{for}_i} \end{aligned}$$

The lock invariant says that the CRDT's state is always the denotation of some set of events $s_{\text{own}} \cup s_{\text{for}}$ (ip is the IP address of replica i). Additionally, the invariant holds resources $\llbracket \frac{1}{3} \rrbracket_{s_{\text{own}}}^{\gamma_{\text{own}_i}} * \llbracket \frac{1}{2} \rrbracket_{s_{\text{for}}}^{\gamma_{\text{for}_i}}$ which guarantee that we are “in sync” with the logical state recorded for replica i in the global invariant **GlobInv**. Notice the lock invariant keeps γ_{for_i} and not γ_{sub_i} because the library knows exactly the set of foreign events that have been processed so far. The table below summarizes where the different fractions of γ_{own_i} , γ_{for_i} and γ_{sub_i} are kept:

	γ_{own_i}	γ_{for_i}	γ_{sub_i}
Global invariant	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{3}$
Lock invariant	$\frac{1}{3}$	$\frac{1}{2}$	
Local state	$\frac{1}{3}$		$\frac{2}{3}$

As part of the proof we also need to define STATELIB's socket protocol $\mathbf{SP}(i, st)$; i.e. a predicate that holds for all states st received by a replica (which dually creates a proof obligation whenever a replica messages others). The abridged version below assumes that st is already deserialized and that i is the replica id of the message's sender:

$$\mathbf{SP}(i, st) \triangleq \exists s'_{\text{own}} s'_{\text{for}}. \text{LocEv}(i, s'_{\text{own}}) * \text{ForEv}(i, s'_{\text{for}}) * \llbracket s'_{\text{own}} \cup s'_{\text{for}} \rrbracket = st \\ * \text{LocStValid}(s'_{\text{own}} \cup s'_{\text{for}}) * \llbracket \text{OM principal}_{\subseteq_{\text{cc}}} (s'_{\text{own}} \cup s'_{\text{for}}) \rrbracket_i^{\gamma_{\text{ccfor}_i}}$$

The socket protocol assumes that the received state st is the denotation of the union $s'_{\text{own}} \cup s'_{\text{for}}$, where s'_{own} and s'_{for} are event sets that are local and foreign, respectively, relative to the message's sender (not relative to the receiver). Additionally, we know that $s'_{\text{own}} \cup s'_{\text{for}}$ is a causally closed subset of the events recorded at replica i (in particular, we know the sender is not accidentally including events that have not been previously recorded).

Since both `apply` and `broadcast` recurse forever, their specifications are not very interesting: in particular, we do not care about their post-conditions. We do care about preserving the lock and global invariants as they execute. We briefly sketch the proof of `apply`. Before `apply` updates the CRDT state via $st := \text{merge } !st \text{ st}'$, we know that all the following hold: the global invariant **GlobInv**, the lock invariant $\mathbf{LI}(i, st)$, and the socket protocol $\mathbf{SP}(j, st')$, where i and j are the ids of the local and sender replicas, respectively, and $i \neq j$. We open the lock invariant and get $!st = \llbracket s_{\text{own}} \cup s_{\text{for}} \rrbracket * \llbracket \frac{1}{3} \rrbracket_{s_{\text{own}}}^{\gamma_{\text{own}_i}} * \llbracket \frac{1}{2} \rrbracket_{s_{\text{for}}}^{\gamma_{\text{for}_i}}$. Similarly, from the socket protocol we know that $st' = \llbracket s'_{\text{own}} \cup s'_{\text{for}} \rrbracket * \text{LocStValid}(s'_{\text{own}} \cup s'_{\text{for}}) * \llbracket \text{OM principal}_{\subseteq_{\text{cc}}} (s'_{\text{own}} \cup s'_{\text{for}}) \rrbracket_i^{\gamma_{\text{ccfor}_i}}$. We would like to apply the coherence lemma (**MergeCoh**), which tells us that merging two states is the same as (1) merging the corresponding events that generated those states, and (2) then taking the denotation of the union of the event sets. The premises of (**MergeCoh**) all follow from local state validity after opening the global invariant, and from the fact that $\llbracket \text{OM principal}_{\subseteq_{\text{cc}}} (s'_{\text{own}} \cup s'_{\text{for}}) \rrbracket_i^{\gamma_{\text{ccfor}_i}}$ proves that the events we are merging have been previously recorded. The resulting logical state is $\llbracket \frac{1}{3} \rrbracket_{s_{\text{own}}}^{\gamma_{\text{own}_i}} * \llbracket (s_{\text{for}} \cup s'_{\text{own}} \cup \{e \in s'_{\text{for}} \mid e.s \neq i\}) \rrbracket_i^{\gamma_{\text{for}_i}}$.

The proof of `get_state` uses the lock invariant to conclude that the returned state is the denotation of the set of events received so far. It then uses the global invariant: specifically the relation $s_{\text{own}} \cup s_{\text{sub}} \subseteq_{\text{cc}} s_{\text{own}} \cup s_{\text{for}}$ in the definition of $\mathbf{LR}(i, s)$ to update the local state to $\text{LocSt}(i, o, f')$. The proof of `update` uses the preservation of global validity under a **TUpdate** transition (Theorem 3) to show that **GlobInv** is preserved. We refer the reader to our formalization for additional details.

8 Verified CRDTs

To test STATELIB we verified five example CRDTs from the literature: grow-only counter (g-counter), grow-only set, product combinator, map combinator, and positive-negative counter (pn-counter). We also verified a closed program consisting of client code that uses the pn-counter. This closed program appears in Nieto et al. [20], and we were able to swap out their op-based pn-counter for our state-based version without modifying their safety proof.¹⁰ We used the closed example as a case study in giving our state-based CRDT the same specification as its op-based counterpart, hiding implementation details in the process.

In this section we focus on describing the pn-counter and the closed program. A pn-counter is a data structure that supports two operations: `add(z)` adds the integer z , which may be negative, to the counter, and `get_value` returns the counter's current value. The initial value is 0. The denotation for pn-counter used in Nieto et al. [20] is $\llbracket s \rrbracket = \sum_{e \in s} e.o.$ We implemented the counter as a wrapper over the product `prod(g-counter, g-counter)`. We now explain how `g-counter` and `prod` work.

The `g-counter` is a simpler version of pn-counter where we can only add non-negative numbers. Operations are of the form `add(n)` with $n \in \mathbb{N}$. If N is the number of CRDT replicas, the lattice state is a vector \vec{c} with N entries. The i th entry tracks the contribution of replica i to the counter's state. The initial value is the 0-vector of length N . The mutator is defined as `mut(\vec{c}, e) = s[e.s \mapsto $\vec{c}_{e.s} + e.o$]`, and merges are done by taking the maximum of two vectors pointwise. The denotation $\llbracket s \rrbracket$ is the pointwise sum of all the vectors in s .

Given two CRDTs $C_A = (\text{Op}_A, \text{St}_A, \text{init}_A, \text{mut}_A, \text{merge}_A)$ and $C_B = (\text{Op}_B, \text{St}_B, \text{init}_B, \text{mut}_B, \text{merge}_B)$, their product is $C_{A \times B} = (\text{Op}_A \times \text{Op}_B, \text{St}_A \times \text{St}_B, \text{init}_A \times \text{init}_B, \text{mut}_{A \times B}, \text{merge}_{A \times B})$. Both the mutator and merge function operate in a component-wise fashion. The denotation $\llbracket s \rrbracket$ splits s into two sets s_A and s_B of A-events and B-events, respectively. It then computes the pair $(\llbracket s_A \rrbracket_A, \llbracket s_B \rrbracket_B)$.

We then implement the pn-counter as a wrapper over the product `prod(g-counter, g-counter)`. Specifically, we wrap the product's `get_state` and `update` as follows (`sum_entries` is the function that sums all the entries of a vector):

```
let pncounter_add z =
  if z >= 0 then prod_update((z, 0))
  else prod_update((0, -z))
let pncounter_get_value () =
  let (v1, v2) = prod_get_state () in
  (sum_entries v1) - (sum_entries v2)
```

Finally, we describe the closed program we verified. This example shows client code interacting with a pn-counter. The relevant snippet is shown below: we have two replicas, A and B , each of which increments the counter, reads it, and then asserts that the read returns one of two possible values. In A the possible values are 1 and 3, depending on whether the remote operation `add 2` has been propagated from B to A by the time the read happens. In either case, the `add 1` must be visible by the subsequent read because the latter happened later according to program order: this is the so-called *reads-follow-writes* session guarantee [8]. An analogous situation happens for replica B .

```
(* replica A *)
add 1;
let r = get_value () in
assert (r = 1 || r = 3)

(* replica B *)
add 2;
let r = get_value () in
assert (r = 2 || r = 3)
```

¹⁰We have added one additional property to the common CRDT resource interface (an excerpt of which was shown in Figure 3), and thus we extended the proof of Nieto et al. [20] such that this additional property is also proved for their op-based library.

The reader can consult Nieto et al. [20] for more details on the proof that the assertions in the example above do not fail, but the important part is that we were able to swap out the op-based counter implementation by our state-based counter, while (almost) keeping the proof intact. The two implementations are quite different: the op-based implementation relies on a causal broadcast algorithm, while ours does not and instead relies on lattice properties, as well as on applications of the product combinator. The fact that both implementations meet the same specification is good evidence that the denotation-based separation logic specifications are general, flexible, and can hide implementation details.

9 Related Work

As previously mentioned, the idea of specifying CRDTs as a partial function from event histories (including causality metadata) to the data type’s state comes from Burckhardt et al. [3]. We also learned from their paper that state-based CRDTs can be thought of as transitively delivering (the effects of) events when states are merged, which in turn makes it possible to prove, as we have done, that state-based CRDTs are causally-consistent.¹¹

Leijnse et al. [13] reformulate the above specification style so it can be better applied to higher-order combinators, coining the term CRDT denotation. They work solely with specifications, and do not implement these combinators nor verify an implementation.

The idea of tracking the state of a concurrent data structure via a logical set of operations that is divided into contributions by the current thread and those originating from other threads, as in our $\text{LocSt}(i, o, f)$ resource, has previously appeared in the context of the FCSL logic (where they are termed “self” and “other” contributions, respectively) [19, 5].

The Compass separation logic framework [4] (also Iris-based) can be used to specify and verify functional correctness of concurrent data structures in a relaxed memory model. There are a number of commonalities with our work: their specs are also given as logically-atomic triples that track the state of a data structure as a function of the set of writes that are visible by a given thread. They develop a notion of logical *views* that is similar to our local snapshots $\text{LocSnap}(i, o, f)$ (without the distinction between own and foreign events): ownership of a view provides a lower bound on the set of observed events, and the views contain logical metadata that tracks the happens-before relation between writes. The main difference with our work is that we operate at the intersection of weak consistency and message passing, whereas their work is in the context of shared memory.

Zeller et al. [24] implement and formally verify multiple state-based CRDTs in Isabelle/HOL. To our knowledge, they are the first to explicitly link denotation-style specifications to their lattice-based implementations. Like us, they prove both convergence and functional correctness. There are two main differences with our work: their system model is an STS where the states map each replica id to the replica state (this is very similar to our STS model from Section 7.1). By contrast, in our work the system model is the operational semantics of *AnerisLang*, with support for mutation, node-local concurrency, higher-order functions, etc. This means that while their technique can only be used to reason about a CRDT in isolation, ours can verify a system where the CRDT and a client (or a larger distributed system of which the CRDT is a small part) are executing together, and where both of these are implemented in a realistic programming language. The second difference is that in Zeller et al.’s work one needs to come up with a different invariant that implies

¹¹They do mention that state-based CRDTs are causally-consistent, but there is no formal proof, or even a precise lemma statement.

coherence between the denotation and the lattice for each implemented CRDT, as well as proving for each example that the invariant is preserved by the different transitions. In our work, we prove just one invariant, global state validity, and the CRDT implementer then needs to prove coherence given that global validity holds.

Nair et al. [18] verify several state-based CRDTs. These are not “pure” CRDTs in the sense that some of data structure’s operations are at times disabled. For example, they show how to verify a distributed lock implemented as a CRDT (the release-the-lock operation can only be enabled if the local replica owns the lock). Proof wise, this means that sometimes it is useful to enforce example-specific invariants on the CRDT being verified: it would be interesting to modify STATELIB so it can support these user-defined invariants and so can tackle the examples presented in their paper.

Gondelman et al. [8] verify functional correctness of a causally-consistent key-value store using *Aneris*. Their work introduces the encoding of causality in separation logic that we use. However, their key-value store is not a CRDT because it violates convergence. Additionally, their work is closer to op-based CRDTs because writes are propagated individually.

Timany et al. [23] develop an extension of *Aneris* called *Trillium* where one simultaneously proves both safety and also that the program being verified refines an STS model. Unlike in our work, where the STS is used just to prove safety, their refinement is *history-preserving*, which allows them to prove liveness properties as well. As one case study, they show that a state-based G-Counter CRDT is eventually consistent. They do not tackle any other (more complex) CRDTs, and their specification of the G-Counter relies on the fact that the G-Counter is monotonic. It would be interesting to recast our work using the *Trillium* methodology with the goal of showing that any CRDT implemented via STATELIB is eventually-consistency (that is, additionally showing eventual delivery).

The closest related work is Nieto et al. [20], from which we inherit the CRDT resource interface from Figure 3, the modular structure of the implementation (a core library that can be instantiated for different CRDT examples), as well as the general structure of the proof: a state-transition system model that is embedded in the logic, as well as a lock invariant that ties the denotation-style specification to the lattice-based state. The main difference is that their paper deals exclusively with op-based CRDTs: as mentioned in Section 4.2 adapting their technique to the state-based setting leads to a number of technical challenges we had to solve in our approach.

10 Conclusions

We have shown how to give modular specifications to realistic state-based CRDT implementations using the *Aneris* separation logic. Our specifications show both convergence and functional correctness relative to an abstract denotational model of the CRDT.

We have explored our approach by implementing and verifying a library, STATELIB, for building state-based CRDTs. Our library takes as input a purely-functional implementation of a state-based CRDT’s core logic, together with coherence proofs between the CRDT’s lattice-based and denotation-based models. The library then produces as output a fully-fledged CRDT that is replicated over multiple nodes. Using the library we implemented and verified multiple example CRDTs from the literature.

When taken together with Nieto et al. [20], our work presents a unified framework for the specification and verification of op- and state-based CRDTs. To show that we can abstract away from the fact that a CRDT is state-based, we re-prove Nieto et al.’s interface for resources tracking CRDT state using a new definition of resources that is compatible with



state-based CRDTs. We test this approach by showing that one can start with a client program that uses an op-based counter CRDT, swap out the counter’s implementation by our state-based implementation, and recover the safety proof for the entire closed program while making minimal changes to the original proof.



References

- 1 Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation log, 2017. URL: <http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf>.
- 2 Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991. doi:10.1145/128738.128742.
- 3 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2014, pages 271–284. ACM, January 2014. doi:10.1145/2535838.2535848.
- 4 Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Thuan Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 792–808, 2022.
- 5 Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent data structures linked in time. *arXiv preprint*, 2016. arXiv:1604.08080.
- 6 Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In *PLDI*, pages 242–255. ACM, 2018.
- 7 Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, 2017. doi:10.1145/3133933.
- 8 Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. Distributed causal memory: Modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi:10.1145/3434323.
- 9 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.
- 10 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.
- 11 Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A mechanised logic for modular reasoning about distributed systems. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 336–365. Springer, 2020.
- 12 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.
- 13 Adriaan Leijnse, Paulo Sérgio Almeida, and Carlos Baquero. Higher-order patterns in replicated data types. In *PaPoC@EuroSys*, pages 5:1–5:6. ACM, 2019. doi:10.1145/3301419.3323971.
- 14 Hongjin Liang and Xinyu Feng. Abstraction for conflict-free replicated data types. In Stephen N. Freund and Eran Yahav, editors, *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 636–650. ACM, 2021. doi:10.1145/3453483.3454067.
- 15 Barbara H. Liskov and Stephen N. Zilles. Programming with abstract data types. In *SIGPLAN Symposium on Very High Level Languages*, pages 50–59. ACM, 1974.

- 16 Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in liquid haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):216:1–216:30, 2020. doi:10.1145/3428284.
- 17 Kartik Nagar and Suresh Jagannathan. Automated parameterized verification of crdts. In *CAV (2)*, volume 11562 of *Lecture Notes in Computer Science*, pages 459–477. Springer, 2019.
- 18 Sreeja S. Nair, Gustavo Petri, and Marc Shapiro. Proving the safety of highly-available distributed objects. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 544–571. Springer, 2020.
- 19 Aleksandar Nanevski. Separation logic and concurrency. oregon programming languages summer school, 2016.
- 20 Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. Modular verification of op-based crdts in separation logic. *Proc. ACM Program. Lang. OOPSLA (2022)*. *Accepted for publication*, 2022.
- 21 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report 7506, INRIA, January 2011. URL: <http://hal.inria.fr/inria-00555588/>.
- 22 Amin Timany and Lars Birkedal. Reasoning about monotonicity in separation logic. In *CPP*, pages 91–104. ACM, 2021.
- 23 Amin Timany, Simon Oddershede Gregersen, Léo Stefanescu, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Unifying refinement and higher-order distributed separation logic. *CoRR*, abs/2109.07863, 2021.
- 24 Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In *FORTE*, volume 8461 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014. doi:10.1007/978-3-662-43613-4_3.

Information Flow Analysis for Detecting Non-Determinism in Blockchain

Luca Olivieri  
University of Verona, Italy
Corvallis Srl, Padova, Italy

Luca Negrini  
Corvallis Srl, Padova, Italy

Vincenzo Arceri  
University of Parma, Italy

Fabio Tagliaferro  
CYS4 Srl, Florence, Italy

Pietro Ferrara  
Ca' Foscari University of Venice, Italy

Agostino Cortesi  
Ca' Foscari University of Venice, Italy

Fausto Spoto  
University of Verona, Italy

Abstract

A mandatory feature for blockchain software, such as smart contracts and decentralized applications, is determinism. In fact, non-deterministic behaviors do not allow blockchain nodes to reach one common consensual state or a deterministic response, which causes the blockchain to be forked, stopped, or to deny services. While domain-specific languages are deterministic by design, general-purpose languages widely used for the development of smart contracts such as Go, provide many sources of non-determinism. However, not all non-deterministic behaviours are critical. In fact, only those that affect the state or the response of the blockchain can cause problems, as other uses (for example, logging) are only observable by the node that executes the application and not by others. Therefore, some frameworks for blockchains, such as Hyperledger Fabric or Cosmos SDK, do not prohibit the use of non-deterministic constructs but leave the programmer the burden of ensuring that the blockchain application is deterministic. In this paper, we present a flow-based approach to detect non-deterministic vulnerabilities which could compromise the blockchain. The analysis is implemented in GoLiSA, a semantics-based static analyzer for Go applications. Our experimental results show that GoLiSA is able to detect all vulnerabilities related to non-determinism on a significant set of applications, with better results than other open-source analyzers for blockchain software written in Go.

2012 ACM Subject Classification Security and privacy → Distributed systems security; Theory of computation → Program analysis; Theory of computation → Program verification; Software and its engineering → Software notations and tools

Keywords and phrases Static Analysis, Program Verification, Non-determinism, Blockchain, Smart contracts, DApps, Go language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.23

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.9.2.23>

Funding *Vincenzo Arceri:* Bando di Ateneo per la ricerca 2022, founded by University of Parma, project number: MUR_DM737_2022_FIL_PROGETTI_B_ARCERI_COFIN, *Formal verification of GPLs blockchain smart contracts*

Pietro Ferrara: SERICS (PE00000014) under the NRRP MUR program funded by the EU – NGEU, iNEST-Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 1.5) NextGeneration EU – Project ID: ECS 00000043, and SPIN-2021 “Static Analysis for Data Scientists” funded by Ca’ Foscari University

Agostino Cortesi: SERICS (PE00000014) under the NRRP MUR program funded by the EU – NGEU, iNEST-Interconnected NordEst Innovation Ecosystem funded by PNRR (Mission 4.2, Investment 1.5) NextGeneration EU – Project ID: ECS 00000043, and SPIN-2021 “Ressa-Rob” funded by Ca’ Foscari University



© Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto Spoto;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 23; pp. 23:1–23:25



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

In the last decade, blockchain software has undergone a notable evolution. In 2008, Bitcoin [33] introduced a Turing-incomplete low-level language to specify locking conditions that must hold for a transaction to be accepted by the network [3]. In 2013, Ethereum [8, 4] provided a Turing-complete bytecode where smart contract rules are enforced by the blockchain consensus. The execution of the code takes place on the Ethereum Virtual Machine (EVM), resulting in software identified as *decentralized applications* (DApps). EVM bytecode is supported by high-level domain-specific languages (DSLs), such as Solidity and Vyper, that have been designed from scratch for the purpose of being executed in the restricted environment of blockchain. Subsequently, thanks to frameworks such as Hyperledger Fabric [2], Tendermint [6, 29], and Cosmos SDK [30], general-purpose programming languages (GPLs) such as Go, Java and JavaScript can also be used to develop smart contracts and DApps, with Go being the most popular in industrial blockchains.

The popularity of GPLs for writing smart contracts and DApps is steadily increasing. Their success is mostly due to the maturity of the languages themselves, directly resulting in wide communities, consolidated tools (such as IDEs and debuggers), and most importantly a pool of expert and knowledgeable developers that can write highly efficient smart contracts. Yet, GPLs were not conceived solely for blockchain ecosystems: code that is harmless and bug-free in other contexts may result in vulnerabilities and errors. Among these, one of the most insidious is non-determinism. When the result of an operation on a blockchain is non-deterministic, there is no guarantee that a common state can be reached by the network's nodes, possibly preventing it from reaching consensus. This can manifest, among other possibilities, as transaction failures or denial of service. Nevertheless, not all instances of non-determinism are intrinsically dangerous: logging the time of a transaction can result in different timestamps appearing in each node's logs, but it does not endanger consensus as it is not *observable* by other nodes. In fact, non-deterministic instructions are problematic only if they can affect the shared blockchain state.

As an example, consider the code in Figure 1, reporting an excerpt of the `ValidateBasic` method from module `x/authz` (part of the Cosmos SDK versions 0.43.x and 0.44.{0,1}) and affected by the vulnerability reported in CVE-2021-41135¹. The code is meant to fail the validation of expired grants. Note that the guard at line 2 involves the local clock of nodes (`time.Now()`) rather than leveraging the timestamp included in the Block header provided by the Byzantine Fault Tolerant clock, that is agreed upon by the consensus. As reported in the official Cosmos forum [12]:

Local clock times are subjective and thus non-deterministic. An attacker could craft many Grants, with different but close expiration times (e.g., separated by a few seconds), and try to exercise the granted functionality for all of them close to their expiration time. It is likely in such a scenario that some nodes would consider a grant to have expired while others would not, leading to a consensus halt.

The code was then fixed in version 0.44.2, but is still a clear example of a vulnerability arising from non-deterministic constructs.

The problem of non-determinism in blockchain software is clearly felt by the communities of the blockchain frameworks treated in this paper. As a representative example, the Tendermint Core documentation [27], while discussing non-determinism, reports:

¹ <https://nvd.nist.gov/vuln/detail/CVE-2021-41135>.

```
1 func (g Grant) ValidateBasic() error {
2     if g.Expiration.Unix() < time.Now().Unix() {
3         return sdkerrors.Wrap(ErrInvalidExpirationTime, "Time can't be in the past")
4     }
5     // [...]
6 }
```

■ **Figure 1** Cosmos SDK code affected by CVE-2021-41135.

While programmers can avoid non-determinism by being careful, it is also possible to create a special linter or static analyzer for each language to check for determinism. In the future we may work with partners to create such tools.

Paper contribution

This paper presents a software verification approach based on static analysis for the detection of non-deterministic vulnerabilities in blockchain ecosystems, covering the most popular frameworks for developing this kind of software, such as Hyperledger Fabric, Tendermint Core and Cosmos SDK. We shift the classical focus that has been applied in this context beyond the mere syntactic absence of non-deterministic constructs. In fact, we aim at distinguishing *harmful* usages of non-determinism, that is, constructs affecting the blockchain state and response, from *harmless* ones. As a consequence, the set of alarms issued to the user sensibly shrinks, as shifting from a syntactic approach towards a semantic one leads to a sensible reduction in false positives. We propose a semantic flow-based static analysis for detecting flows from non-deterministic constructs to blockchain state modifiers and response builders. The choice of a flow-based analysis seems natural when the problem is phrased as “*is there execution where a non-deterministic value affects the blockchain state or the contract’s response?*”. We thus exploit the well-consolidated literature in this area to adopt scalable solutions that soundly over-approximate all program executions.

We provide a static analyzer implementing our approach: GoLiSA², a sound static analyzer based on abstract interpretation [10] for Go applications. Intuitively, we use our analyzer’s fixpoint engine to mark *all* program variables (local variables, objects’ fields, ...) that can contain values affected, directly or indirectly, by a non-deterministic construct or computation. Specifically, we can perform a shallower analysis detecting only explicit flows using *Taint* analysis [43, 14], where non-deterministic constructs and blockchain state modifiers are modeled as sources and sinks, respectively. Alternatively, we can perform a deeper analysis able to also detect implicit flows by means of the *Non-interference* analysis [24, 25], where non-deterministic constructs and blockchain state modifiers are instead modeled as low and high variables, respectively. Both solutions are implemented in GoLiSA, whose analysis starts by syntactically visiting the input application to annotate all sources and sinks. The annotations are dynamically generated depending on the kind of application of interest (i.e., Hyperledger Fabric, Cosmos SDK, or Tendermint Core). Since there is no predefined set of sources in the target program, both *Taint* analysis and *Non-interference* are parametric: they consider as *harmful* (i.e., tainted or low integrity, depending on the analysis that is to be executed) only variables that are annotated as sources. The fixpoint engine then takes care of propagating values coming from sources on the entirety of the program, exploiting our analyses implementations. After the fixpoint converges, a mapping stating if each program variable is the result of a non-deterministic computation is available at each program

² <https://github.com/lisa-analyzer/go-lisa>

point. These are then used by our non-deterministic semantic checkers, that visit the whole application searching for statements annotated with the sink annotation. Whenever one is found, the mappings are used to determine if the values used as parameters of the call are critical or, in the case of *Non-interference*, if the call happens on a critical state.

Our approach, as highlighted by our evaluation, shows a significant decrease of false positives on real-world blockchain applications compared to other analyzers for blockchain non-determinism. The solution has been experimented on a benchmark of more than 600 real-world blockchain programs written in Go. These show that GoLiSA is able to perform the analysis on the totality of smart contracts in this significant benchmark, and to successfully report their non-determinism vulnerabilities.

The analyses are then evaluated in terms of precision of the results (true positive, false positive, and false negative alarms). Based on these criteria, GoLiSA outperforms existing open-source static analyzers for Go blockchain software. Moreover, the evaluation shows that the execution time of the analyses is not impractical for real use cases.

To the best of our knowledge, GoLiSA is the first sound semantic-based static analyzer for blockchain software able to precisely detect critical non-determinism behaviors while scaling to real-world programs.

Summarizing, our contribution is threefold, as we provide:

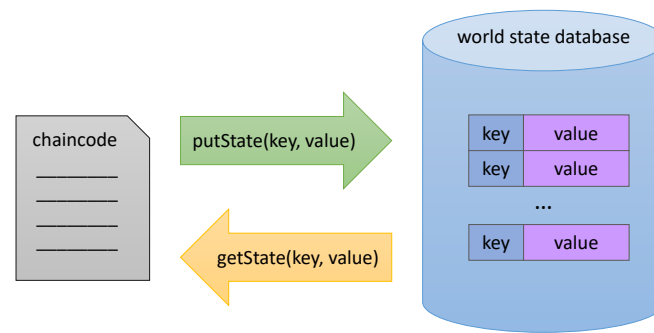
- a detailed investigation on the sources and the sinks that lead to non-determinism issues in the most popular blockchain frameworks;
- a flow-based static analysis for the detection of critical non-determinism behaviors, with two instantiations exploiting different formalizations;
- an open-source sound static analyzer for detecting critical non-deterministic behaviors in blockchain software written in Go.

Paper structure

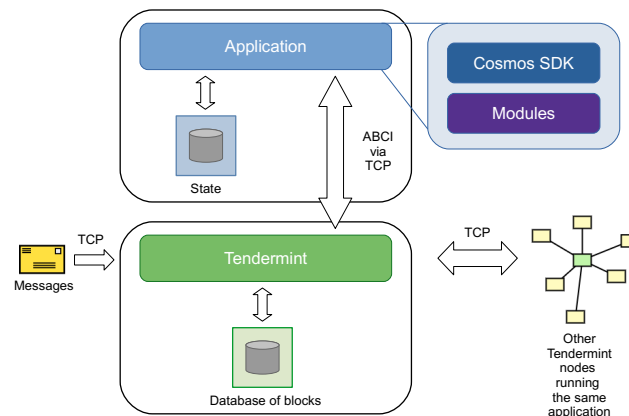
Sect. 2 reports an overview about blockchain software using Go and the most popular frameworks to develop it. Sect. 3 discusses the problem of non-deterministic behavior in blockchain context. After reporting an overview on information flow analyses, Sect. 4 presents our core contribution for detecting non-deterministic behavior in blockchain software, that relies on GoLiSA. Sect. 5 reports our experimental results. Sect. 6 discusses the related work. Finally, Sect. 7 concludes the paper.

2 Preliminaries: Go and Blockchain

Go (<https://golang.org>) is a statically typed, compiled, open-source, and general-purpose high-level programming language designed by Google to speed up software development, and that is appreciated for its cross-compilation feature. Its versatility and performance contributed to its diffusion in the blockchain environment: popular frameworks such as the Hyperledger Fabric³, Tendermint⁴ and the Cosmos SDK⁵ are written in Go. These rely on Go to develop efficient smart contracts and DApps, exploiting its high performances.



■ **Figure 2** The world state database of Hyperledger Fabric.



■ **Figure 3** Cosmos SDK architecture.

2.1 Blockchain Environments

Hyperledger Fabric (HF) is a permissioned blockchain framework designed to be adopted in enterprise contexts, supported by the Linux Foundation and other contributors such as IBM, Cisco, and Intel. In HF, smart contracts and DApps are written in *chaincode* that can be implemented in several GPLs such as Go, JavaScript, and Java. In most cases, the chaincode interacts only with the world state database component of the ledger, and not with the transaction log [26]. Go is currently the most popular language on GitHub related to *chaincode*⁶, as Go smart contracts are the best performing ones [23].

Tendermint Core, recently rebranded as *Ignite*, is a platform for building blockchain nodes, supporting both public and permissioned *proof-of-stake* (PoS) networks. It is a Byzantine Fault Tolerant (BFT) middleware that separates the application logic from the consensus and networking layers, allowing one to develop blockchain applications written in any programming language, including Go, and replicate them on many machines [7].

Cosmos SDK is an open-source Go framework that eases the development of blockchain applications while optimizing their execution by running them on Tendermint Core. As shown in Figure 3, Cosmos SDK abstracts all the boilerplate code needed to set up a Tendermint

³ <https://www.hyperledger.org/use/fabric>

⁴ <https://tendermint.com/>

⁵ <https://v1.cosmos.network/sdk>

⁶ Querying the keyword `chaincode` on GitHub (<https://github.com/search?q=chaincode>) results in more than 2100 repositories, about half of which are written in Go. Accessed: 01-12-2022.

Core node, allowing for customized protocol configurations. The programming style follows the object-capability model, where the security of subcomponents is imperative, especially those belonging to the core library. Cosmos SDK is a framework for DApps, supporting different functionalities through highly customizable modules (that can also manage smart contracts).

2.2 Blockchain Consensus

Consensus protocols ensure the validity and authenticity of transactions performed in the blockchain, as they check results of smart contracts or DApps computations through the state of the network's nodes. If a given number of nodes agree on the final state, consensus is reached and the transaction is validated. Otherwise, it is discarded and the nodes proposing spurious states are excluded from the network. When consensus cannot be reached, the blockchain either forks or halts. Deterministic execution is thus required for software that runs in a blockchain, as it guarantees that, when starting from a common state, the same result is reached in any distinct blockchain node, avoiding inconsistencies among peers and consensus failures. Nevertheless, GPLs provide several components that can explicitly lead to non-determinism, such as (pseudo-)random number generators or external computations. Furthermore, even methods that are explicitly sequential and deterministic pose a threat when executed on different nodes, such as the `time.Now()` call from Figure 1. Despite these threats, popular blockchain frameworks such as HF and Cosmos SDK do not enforce particular restrictions on the usage of non-deterministic methods and components.

3 Non-Deterministic Behaviors in Blockchain Software: Sources and Sinks

When trying to prevent non-deterministic vulnerabilities, a first solution is to limit the expressiveness of the GPL by either black- or white-listing APIs and constructs. Consider the Go snippets reported in Figure 4. Both fragments rely on the `time` API to retrieve a timestamp from the host system. In general, the results of calls to the `time` API are subjective to the node executing them, and they might lead to blockchain non-determinism due to different system settings (e.g., time, date, time zones, ...) or due to nodes executing the code at slightly different times. Specifically, Figure 4a shows a safe usage of the `time` API: the timestamp is only used for logging with no observable consequences on the blockchain state or the execution result. Instead, Figure 4b reports a problematic usage of the API, as the timestamp is stored in the blockchain using `PutState`, an HF-specific function that updates the shared network state. Since timestamps could differ on each node, this potentially leads to inconsistent executions (i.e., different blockchain states or execution results), causing transaction failure⁷.

It should thus be evident that identifying sources of non-determinism and preventing their usage is not enough when we aim at discerning between harmful and harmless non-deterministic constructs. In fact, one should also recognize how these are used, determining if they can influence the shared blockchain state. In the rest of this section we discuss, for each blockchain framework presented in Section 2, (i) the constructs that generate potentially harmful non-determinism (that is, *sources* of non-deterministic values), and (ii) the blockchain

⁷ In this case, the `GetTxTimestamp` method from the HF API should have been used instead of `time.Now`.

```

1 func transfer(from, to Address, value int64, stub *shim.ChaincodeStub) {
2     start := time.Now()
3     //... transfer operations that takes some milliseconds ...
4     elapsed := time.Now().Sub(start)
5     log.Println("Time elapsed for the transfer operations: ", elapsed)
6 }

```

(a) Example of safe use of the time API.

```

1 func transfer(from, to Address, value int64, stub *shim.ChaincodeStub) {
2     t := time.Now()
3     //... transfer operations ...
4     err := shim.PutState("transaction-time", t)
5     //... other operations ...
6 }

```

(b) Example of issue of non-determinism with the time API.

■ **Figure 4** Examples of harmless and harmful non-determinism in blockchain.

state modifiers and response builders (i.e., statements that make a transaction succeed or fail), namely *sinks* that are sensitive to non-determinism⁸. This will prepare the ground for the core contribution of this paper: a static approach to detect critical usages of non-determinism in blockchain software, reported in Section 4.

3.1 Sources of Non-Determinism

The sources of non-determinism can be logically split in two families, the first being related to the combination of framework and GPL adopted to develop the software. This family comprises a set of constructs and APIs allowed by the framework that may break the consensus during the execution of smart contracts or DApps. In Go, these are:

- *iteration over maps* that, being the iteration order unspecified⁹, is not guaranteed to be deterministic;
- *parallelization and concurrency*, that can lead to race conditions on shared resources, thus creating non-determinism on the computed values;
- *global variables*, that may change innately and cause inconsistencies to the results, since they depend on the application state of a peer and not on that of the blockchain [32, 5].
- *random value generators*, that can potentially be allowed in smart contracts [9] to employ custom logic while being non-deterministic by-definition.

The second family instead involves statements related to the underlying environment, such as file systems, operating systems, databases, and Internet connections. While these are not intrinsically non-deterministic, they become dangerous when their result is expected to be consistent on different environments. These comprise APIs handling:

- *file systems*, as the program might rely on files that are not present on all nodes, as they might have been deleted, edited, moved, or there might be insufficient disk space causing any operation to fail;
- *operating systems* (OS), since the blockchain might operate on various hosts and language APIs could return different results on each OS (e.g., time and date methods could return different values if nodes are not synchronized);

⁸ The complete list of sources and sinks of non-determinism is available at <https://github.com/lisa-analyzer/go-lisa/blob/master/go-lisa/sources-sinks.md>.

⁹ https://golang.org/ref/spec#For_statements

■ **Table 1** Potential non-deterministic behaviors related to Go.

<i>Level</i>	<i>Category</i>	<i>Package</i>	<i>Statements/Methods</i>
<i>Framework/Language</i>	Map iteration	-	<code>range</code> on map
	Parallelization/concurrency	-	<code>go</code> (Go routine), <code><-</code> (channel)
	Random number generation APIs	<code>math/rand</code> , <code>crypto/rand</code>	*
	Global variables	-	-
<i>Environment</i>	File system APIs	<code>io</code> , <code>embed</code> , <code>archive</code> , <code>compress</code>	*
	OS APIs	<code>os</code> , <code>syscall</code> , <code>internal</code> , <code>time</code>	*
	Database APIs	<code>database</code>	*
	Internet APIs	<code>net</code>	*

```

1 func (s *SmartContract) transaction(APIStub shim.ChaincodeStubInterface) sc.Response {
2
3     if rand.Int() % 2 == 0 {
4         return shim.Error("Fail")
5     } else {
6         return shim.Success(nil)
7     }
8 }

```

■ **Figure 5** Example of issue of non-determinism related to the blockchain response.

- *databases*, where records might be deleted, edited, or contain different data;
- *Internet connections*, as networking setup or errors could cause some addresses to be unreachable on few nodes of the network.

Table 1 summarizes the instructions and libraries of Go¹⁰ that we considered as cases of non-determinism, where * represents the entirety of the package. For the sake of simplicity, the table reports instructions and packages omitting the full signatures of each method. Note that only few methods within those packages lead to non-deterministic behaviors: for instance, most methods from package `time` handling dates and times do not pose a threat in smart contracts and DApps, and are in fact quite common. However, operations such as retrieving the current time of the OS (i.e., methods `Since`, `Now`, `Until`) are potentially dangerous.

3.2 Sinks of Non-Determinism

Sinks of non-determinism comprise constructs and APIs with the ability of both modifying the common state of the blockchain or having an impact on the response of blockchain networks. While the former is inherently involved in consensus protocols, the execution of code within the blockchain does not necessarily change the shared state (e.g., functions that simply read a value). However, the execution may lead to non-deterministic responses, compromising the consensus of the network, as in the simple example reported in Figure 5. Table 2, where the **Critical point** column identifies what part of the API should not receive non-deterministic values, summarizes the main instructions and components that we consider as sinks for non-determinism.

3.2.1 Hyperledger Fabric APIs for Go

In HF, chaincode executes transaction proposals against world state data that may change its state. Programmatically, interface `ChaincodeStubInterface` from the HF Go APIs enables access and modification of the blockchain state. Table 2 reports the current components (as

¹⁰The full list of Go APIs sources considered in our analyses is available at https://github.com/lisa-analyzer/go-lisa/blob/master/go-lisa/src/main/resources/for-analysis/nondeterm_sources.txt. The list consider API until Go version 1.17.

■ **Table 2** Main sinks for blockchain software written in Go.

Framework	Package	Type/Interface	Statements/Methods	Critical point
<i>HyperLedger Fabric</i>	shim	ChaincodeStubInterface	PutState DelState PutPrivateData DelPrivateData Success Error	parameters parameters parameters parameters statement statement
<i>Tendermint Core</i>	abci/types	Application	ResponseBeginBlock ResponseDeliverTx ResponseEndBlock ResponseCommit ResponseCheckTx	instance returned instance returned instance returned instance returned
<i>Cosmos SDK</i>	types	KVStore	Set Delete	parameters parameters
	kv, dbadapter, gaskv, iavl, listenkv, prefix, tracekv,	Store	Set Delete	parameters parameters
	types/errors		ABCIError Redact ResponseDeliverTx ResponseCheckTx WithType Wrap Wrapf	statement statement statement statement statement statement

of version 2.4) involved in the data-write proposal. The semantics of these components does not affect the blockchain state until the transaction is validated and successfully committed. Hence, if these components lead to different results (i.e., changes to the shared state) due to non-determinism, consensus will not validate the transaction and no new state will be committed. Regarding the response statements, HF provides the **Success** and **Error** methods to yield successful and failed transaction responses, respectively.

3.2.2 Tendermint Core APIs for Go

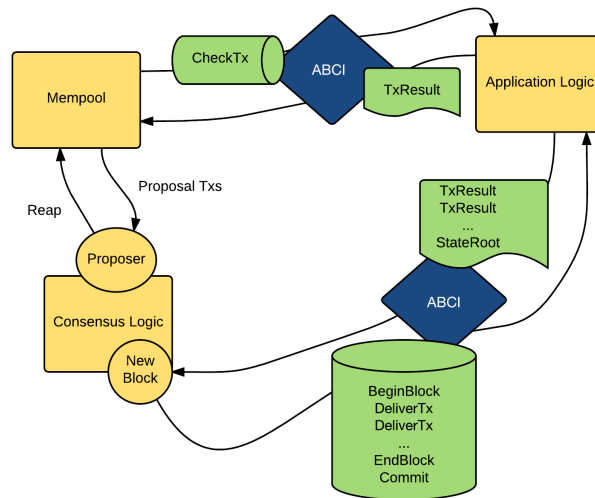
Tendermint Core is a middleware with no explicit access to application state by design, enabling communication through the *Application Blockchain Interface* (ABCI¹¹). Figure 6 depicts the consensus process used to validate and store a transaction using the ABCI methods. As reported in the official documentation [27] of Tendermint Core v. 0.35.1, only **BeginBlock**, **DeliverTx**, **EndBlock**, and **Commit** must be strictly deterministic to ensure consensus. Although the logic of these methods is different, they possess similar structure: they all accept a request and return a response (**ResponseBeginBlock**, **ResponseDeliverTx**, **ResponseEndBlock**, **ResponseCommit**), with the latter that must be deterministic.

3.2.3 Cosmos SDK APIs

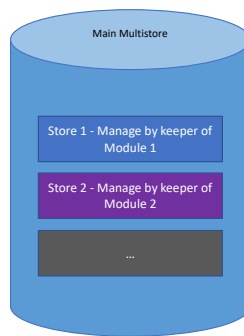
Cosmos SDK handles both the application and the blockchain state through the *store*¹². At a high level, the store is a set of key-value pairs used to store and retrieve data, implemented by default as a *multistore* (i.e., a store of stores), as shown in Figure 7. The multistore encapsulation enables modularity of the Cosmos SDK, as each module declares and manages

¹¹ <https://github.com/tendermint/tendermint/blob/7983f9cc36c31e140e46ae5cb00ed39f637ef283/docs/introduction/what-is-tendermint.md#abci-overview>.

¹² <https://github.com/cosmos/cosmos-sdk/blob/2b24afad075894dd1727d057f87e2be24238016f/docs/core/store.md>.



■ **Figure 6** ABCI methods and consensus flow.



■ **Figure 7** Main store of Cosmos SDK.

its own subset of the state using specific keys. Keys are typically held by *keepers*, a Cosmos SDK abstraction with the role of managing access to the multistore's subset defined by each module. The `Store` type is declared in several packages (e.g., `kv`, `tracekv`, `gaskv`, `ival`), with all definitions implementing the `KVStore` interface. The latter provides common APIs to access and modify the state of the blockchain using methods such as `Set` and `Del`. As for responses, Cosmos provides several methods (such as `ABCIError`, `Wrap`, `ResponseDeliverTx`) in package `types/errors` to return failed transaction responses.

4 Information Flow Analysis for Non-Determinism Detection

In this section we introduce and discuss our approach for detecting non-deterministic behaviors in blockchain software. In particular, we consider non-determinism as *critical* only if a non-deterministic value can affect the blockchain state, either directly (i.e., being stored inside the state) or indirectly (e.g., guarding the execution of state updates). Any other usage of non-determinism is considered safe, as it does not affect the blockchain state or response. As such, when mentioning non-determinism in the remainder of the chapter, we implicitly refer

<pre> 1 var l, h 2 h := 1 </pre>	<pre> 1 var l, h 2 if l = true then 3 h := 3 4 else 5 h := 42 </pre>	<pre> 1 var l, h 2 if h = 1 then 3 (* time-consuming work *) 4 l := 0 </pre>
(a)	(b)	(c)

■ **Figure 8** Example of (a) explicit, (b) implicit, and (c) side channel flows, where `h` and `l` represent secret and public variables respectively.

to its critical version. We rely on information flow analysis for detecting values originating from sources of non-determinism that can affect the state of the blockchain. We only focus on static analyses, since they soundly over-approximate all possible behaviors of target programs and can thus give guarantees about the absence of such behaviors. We instantiate two types of analyses: a *Taint* analysis, able to capture the so-called *explicit flows*, and a *Non-interference* analysis, that can also detect *implicit flows*.

4.1 An Overview on Information Flow

Information flow analyses [11, 38] address the problem of understanding how information *flows* from one variable to another during a program’s execution. These analyses usually partition the space of program variables into *private* (or *secret*) and *public*, with the latter being accessible to – and in some cases also modifiable by – an external attacker. The goal of these analyses is then to find program executions where information *flows* from one partition to the other, that is, where values of variables from one partition can affect the values of variables from the other one. Figure 8 reports examples¹³ of the three main types of flows, namely:

- *explicit flow*: when a secret variable is assigned to a value obtained starting from public variables;
- *implicit flow*: when an assignment to a secret variable is conditionally executed depending on values of public variables;
- *side channel*: where some observable properties of the execution, e.g., the amount of computational resources used, depends on the values of some secret variables.

In general, the term *source* is traditionally used for variables holding values that one wants to track along program executions, while *sink* is used to describe locations where values coming from sources should not flow. Using this terminology, when the property of interest ensures the *integrity* of secret variables, information flow analyses can be instantiated using public variables as sources and private ones as sinks, exactly as in Fig. 8 and in the list above. These are able to detect situations where (i) a possibly corrupted value provided by a malicious attacker could be stored into variables whose content is supposed to be reliable, or (ii) such a value governs the update to private variables. If, however, one wants to ensure the *confidentiality* of secret variables, the same analyses can be recasted with private variables acting as sources and public ones as sinks, thus searching for flows in the opposite direction. The target of the analysis is then to find disclosures of private data to external entities.

In the context of non-deterministic behaviors in blockchain environments, information flow analyses can be used to detect when non-deterministic values end up or affect the blockchain’s state, thus checking the *integrity* of that state w.r.t. non-deterministic values.

¹³[https://en.wikipedia.org/wiki/Information_flow_\(information_theory\)](https://en.wikipedia.org/wiki/Information_flow_(information_theory)).

As such, we are interested in information flowing from public to private variables, and we will use *sources* to identify ones that are initialized to non-deterministic values and *sinks* to identify all variables that have an effect on the blockchain’s state. Moreover, we will focus on explicit and implicit flows. In fact, side channels are typically studied to detect secret information leaking through, for instance, execution time, thus violating the confidentiality of that information instead of its integrity. On the other hand, explicit and implicit flows identify non-deterministic values that are either used to update the blockchain’s state or a transaction’s result, or that govern their execution. As a concrete example, recall the code from Figure 1: the vulnerability presented there is an implicit flow since the blockchain’s state is not directly updated with non-deterministic values, but the execution of the update (i.e., the `return` statement) is conditional to some non-deterministic value (i.e., `g.Expiration.Unix() < time.Now().Unix()`).

In the following, we introduce two well-established information flow analyses that we will use for non-determinism detection.

4.1.1 Non-Interference

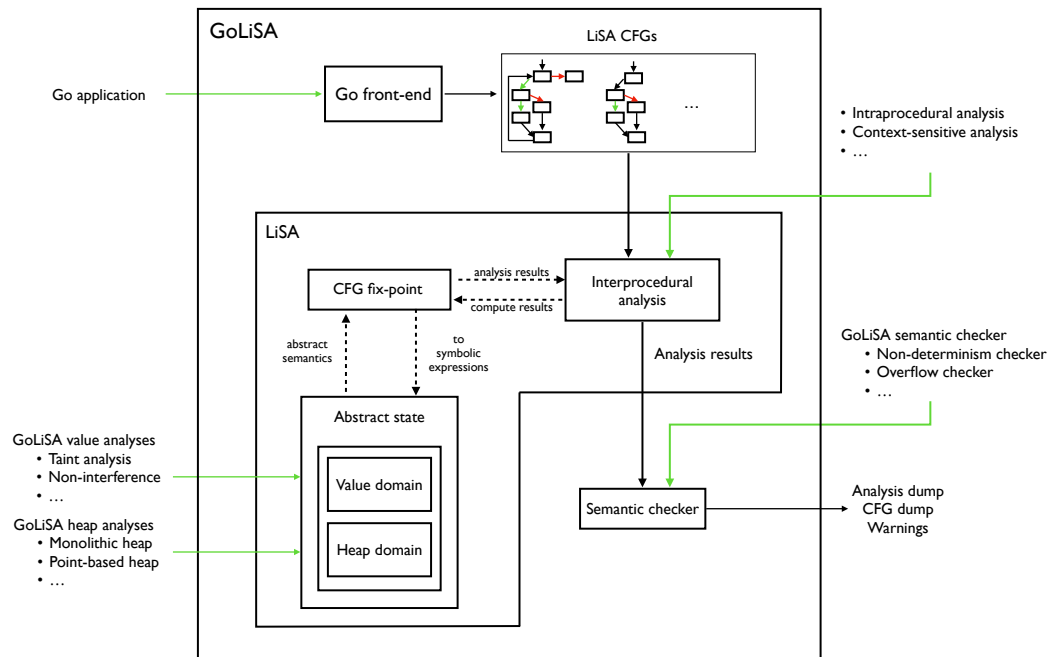
Non-interference [24, 25] is a notion of security capturing the idea that if computations over private information are independent from public information, then no leakage of the former can happen. In simple terms, after partitioning the space of inputs of a program P into *low* (private or secret, denoted by \mathbb{L}), and *high* (public or available to anyone, denoted by \mathbb{H}), *Non-interference* is satisfied if changes in the high input do not affect the observable (i.e., low) output of the program:

$$\forall i_{\mathbb{L}} \in \mathbb{L}, \forall i_{\mathbb{H}}, i'_{\mathbb{H}} \in \mathbb{H} . P(i_{\mathbb{L}}, i_{\mathbb{H}})_{\mathbb{L}} = P(i_{\mathbb{L}}, i'_{\mathbb{H}})_{\mathbb{L}}$$

This notion is often instantiated in language-based security by partitioning the space of program variables between \mathbb{L} and \mathbb{H} , and finding instances of explicit or implicit flows between these partitions. Such analysis computes, for each program point, a mapping from variables to the information level they hold (low or high), while also keeping track of an execution state depending on the information level of the Boolean conditions that guard the program point. Violations of *Non-interference* for integrity can then be detected whenever an assignment to a variable in \mathbb{H} either (i) assigns a low value (that is, an expression involving variables in \mathbb{L}), or (ii) happens with a low execution state (that is, guarded by at least a Boolean condition that involves variables in \mathbb{L}), thus identifying both explicit and implicit flows. This can be formalized as a type system for security [38].

4.1.2 Taint Analysis

Taint analysis [43, 14] is an instance of information flow analysis that can be seen as simplification of *Non-interference* considering only explicit flows. In this context, variables are partitioned into *tainted* and *untainted* (or *clean*), with the former representing variables that can be tampered with by an attacker and the latter representing variables that should not contain tainted values across all possible program executions. Roughly, *Taint* analysis corresponds to the language-based *Non-interference* instantiation without the execution state, thus unable to detect implicit flows. *Taint* has been instantiated to detect many defects in real-world software, such as web-application vulnerabilities [16], privacy issues [22] (also related to GDPR compliance [20]), and vulnerabilities of IoT software [17].



■ **Figure 9** GoLiSA overall execution.

4.2 The GoLiSA Static Analyzer

GoLiSA¹⁴ is an abstract interpretation [10] based static analyzer for Go applications, on which we will rely for the rest of the paper for reasoning about blockchain software written in Go. In this section, we present its architecture and its main feature. GoLiSA relies on LiSA [19, 34] (Library for Static Analysis¹⁵), a Java library that provides a complete infrastructure for the development of static analyzers based on abstract interpretation. In particular, LiSA implements several standard components of abstract interpretation-based analyzers, such as an extensible control-flow graph representation (CFG), a common analysis framework for the development of new static analyses, and fixpoint algorithms on LiSA CFGs.

The high-level analysis process of GoLiSA is reported in Fig. 9. The analysis starts with the *Go front-end* (a sub-component of GoLiSA) that compiles Go source code into LiSA CFGs and defines the semantics, types and language-specific algorithms that implement the Go execution model, capturing the peculiarities of Go in order to make them understandable to LiSA (e.g., scoping and shadowing of variables¹⁶). These CFGs are then passed to LiSA, that analyzes them in a generic language-independent fashion. Roughly, CFGs are passed to an *interprocedural analysis*, a component that cooperates with a *call graph* to resolve calls and compute their results. The interprocedural analysis computes fixpoints over CFGs according to some implementation-specific logic (e.g., modularly, relying on call chains, ...). Each individual fixpoint relies on language-specific analysis-independent semantics for CFG nodes, that is directly provided by GoLiSA: each node is rewritten into a sequence of *symbolic expressions*, modelling the effects that executing a high-level instruction has on the program state through low-level atomic semantic operations. Each of these symbolic expressions is fed

¹⁴ Available at <https://github.com/lisa-analyzer/go-lisa>

¹⁵ LiSA project and documentation available at <https://github.com/lisa-analyzer/lisa>

¹⁶ https://go.dev/ref/spec#Declarations_and_scope

to an *abstract state* [15], a combination of an abstract domain modelling the dynamic memory of the program (*heap domain*, e.g., point-based heap analysis [1]) and one for tracking values of program variables and memory locations (*value domain*, e.g., intervals [10]). The abstract state and its underlying domains compute a sound over-approximation of the expression's effects according to their specific logic, and this can later be exploited by *semantic checks* to issue warnings that are of interest for the user. All analysis components (interprocedural analysis, call graph, abstract state, heap domain, value domain and semantic checks) are part of LiSA's configuration, enabling modular composition and implementation of each component.

4.3 GoLiSA for Non-Deterministic Behaviors Detection

At this point, we are in position to instantiate GoLiSA for the static detection of non-deterministic behaviors in blockchain software. The core idea of our solution is to track the values generated by the hotspots identified in Section 3.1 during the execution of a program using either *Taint* analysis or *Non-interference*. Similarly, after the analysis completes, we can use a semantic checker to exploit the abstract information provided by the domain of choice, checking if any of the sinks specified in Section 3.2 receives one such non-deterministic value as parameter or, in the case of *Non-interference*, if the sink is found in a *low* execution state.

GoLiSA's analysis is instantiated as follows:

- *Taint* analysis and *Non-interference* are implemented as value domains, both of them being non-relational domains (i.e., mapping from variables to abstract values – taintedness and integrity level respectively – with no relations between different variables), with *Non-interference* keeping track of the abstractions for each guard;
- field-insensitive program point-based heap domain (Section 8.3.4 of [37]), where any concrete heap location allocated at a specific program point is abstracted to a single abstract heap identifier;
- context-sensitive [39, 28] interprocedural analysis, abstracting full call-chain results until a recursion is found;
- runtime types-based call graph, using the runtime types of call receivers to determine their targets;
- two semantic checkers, for *Taint* analysis and *Non-interference*, that scan the code in search for sinks, checking the taintedness or integrity level of each sink.

The analysis begins by visiting the input program to detect the statements annotated as sources and propagating the information from them. The analyses produce, for each program point, a mapping stating if each variable is the result of a non-deterministic computation. These mappings are then used by our semantic checkers, that visit the program in search for statements annotated as sinks. When one is found, the mappings are used to determine if values used as parameters of the call are critical or, in the case of *Non-interference*, if the call happens on a critical state. The choice of the analysis to run (and thus of the checker to execute) is left to the user.

For instance, let us consider the fragment reported in Figure 4a. At line 5, despite variable `elapsed` being marked as tainted, no warning is raised by GoLiSA regardless of the chosen analysis, as it does not reach any sensitive sink. Instead, analyzing the fragment from Figure 4b results in the following alarm:

```
The value passed for the 2nd parameter of this call is tainted,
and it reaches the sink at parameter 'value'
```

The warning is issued with both analyses, since variable `t` is marked as tainted and reaches a blockchain state modifier through an explicit flow.

Consider now the example reported in Figure 1. Here, no explicit flow happens at line 3, that contains the blockchain state modifier `Wrap`, but its execution depends on the non-deterministic value used in the condition at line 2, that is, `time.Now().Unix()`. As this is an implicit flow, the *Taint* analysis is not able to detect it. GoLiSA will however discover it with *Non-interference*, raising the following alarm:

```
The execution of this call is guarded by a tainted condition,
resulting in an implicit flow
```

4.4 Detection of Sources and Sinks in GoLiSA

To exploit information flow analyses, the analyzer must know which are the sources and sinks of the program. In this regard, GoLiSA provides a solution based on annotations, marking the corresponding statements as sources and sinks. In the following, we describe how GoLiSA annotates sources (Table 1) and sinks (Table 2) depending on their types.

Methods and functions

As shown in Tables 1 and 2, all sinks and several sources correspond to functions and methods of APIs from either the Go runtime or the blockchain frameworks. GoLiSA contains a list of the signature of these functions and methods and it automatically annotates the corresponding calls in the program by syntactically matching them. While we rely on manual annotations, they can also be generated using automated tools (e.g., SARL [18]). For instance, when GoLiSA iterates over the following snippet, it is able to discover the call to `time.Now`, that gets annotated as source, and the one to `PutState`, whose parameters get annotated as sinks:

```
1 key := "key"
2 tm := time.Now()
3 stub.PutState(key, []byte(tm))
```

Then, the information flow analysis propagates taintedness from the return value of `time.Now` to the second parameter of `PutState`, thus issuing an alarm at line 3.

Map iterations

To detect iterations over maps, one needs to reason about typing. GoLiSA exploits runtime types inferred by the analysis to identify `range` statements happening over maps. If a map iteration occurs, that is, if the object in a `range` statement is inferred to be a map, then GoLiSA marks as sources the variables used to store keys and values of the map. Consider as an example the following code snippet:

```
1 s := ""
2 kvs := map[string]string{"a": "hello", "b": "world!"}
3 for k, v := range kvs {
4     s += v
5 }
6 stub.PutState("key", []byte(s))
```

While analyzing the code, `range` statements are checked for the types of their parameter. GoLiSA annotates as sources both `k` and `v`, as `kvs` is inferred to be a map, while the sink at line 6 is detected through already discussed annotations. Information flow analyses can then propagate the taintedness from `v` to `s`, that in turn flows to the second parameter of `PutState`, issuing an alarm at line 6.

23:16 Information Flow Analysis for Detecting Non-Determinism in Blockchain

Global variables

GoLiSA syntactically annotates every global variable appearing in the program as a source of non-determinism, as their value could be modified independently on each peer. For instance, in the following code, the value of global variable `glob` could differ from peer to peer depending on the number of times function `inc` has been executed. This can happen as not all peers simulate the same transaction, for instance due to differences in the endorsement policy of each peer [32].

```
1 var glob string
2 func inc() {
3     glob += "a"
4 }
5 func (s *SmartContract) transaction(stub shim.ChaincodeStubInterface) sc.Response {
6     stub.PutState("key", []byte(glob))
7 }
```

Before the analysis, GoLiSA iterates over all program components, annotating `glob` as a source. The sink at line 6 is annotated as sink as previously discussed. Then, the information flow analysis propagates taintedness from `glob` to the second parameter of the call to `PutState`, raising an alarm at line 6.

Go routines

GoLiSA inspects the code of Go routines, checking the scope of variables they use. If these are defined outside the routine using them, they are effectively shared among threads, potentially leading to race conditions or non-deterministic behaviors. Hence, GoLiSA annotates the such variables as sources. As an example, the following snippet defines and invokes a simple Go routine that modifies a variable defined in an enclosing scope:

```
1 s := ""
2 go func(){
3     for i := 1; i <= 10000; i++ {
4         s += "0"
5     }
6 }
7 stub.PutState("key", []byte(s))
```

When GoLiSA finds the Go routine, it checks the scopes of each variable, inferring that `s` is declared outside the routine itself. Hence, GoLiSA annotates `s` at line 1 as source, while the sink at line 7 is annotated as previously discussed. Then, the information flow analysis propagates taintedness from `s` to the second parameter of `PutState`, issuing an alarm at line 7 since the value of `s` depends how many times the Go routine has executed the loop body.

Go channels

Channels are pipes that connect concurrent Go routines. Operator `<-` allows interaction with channels to retrieve a value from them, blocking until one is available. GoLiSA annotates as sources the instructions reading values from channels, as the order in which these are written is intrinsically non-deterministic. Consider the following example:

```
1 c := make(chan int)
2 go myroutine1(c)
3 go myroutine2(c)
4 x, y := <- c, <- c
5 stub.PutState("key", []byte(x))
```

GoLiSA iterates over the program searching for occurrences of the operator `<-`. It then annotates variables `x` and `y` as sources, as they receive a value from channel `c`. The sink at line 5 is detected as previously discussed. The information flow analysis can then propagate taintedness from `x` to the second parameter of `PutState`, resulting in an alarm at line 5.

5 Experimental Evaluation

In this section, we discuss the experimental evaluation of the information flow analyses implemented in GoLiSA to detect non-determinism issues in real-world blockchain software. First, we study them from a quantitative point of view, on a set of 651 real-world HF smart contracts retrieved from public GitHub repositories. The evaluation focuses on the HF framework since, to the best of our knowledge, it is the only framework supported by several static analyzers detecting non-determinism issues. This will allow us to compare GoLiSA against state-of-the-art static analyzers in this domain. Furthermore, HF is currently the most popular and widespread blockchain framework among public GitHub repositories, with most smart contracts written in Go. Nevertheless, GoLiSA provides support also for detecting non-determinism behaviors for Cosmos SDK and Tendermint Core smart contracts and DApps.¹⁷

We compare GoLiSA with two open-source static analyzers for chaincodes, namely `Reve^CC` and `ChainCode Analyzer`. The experiments show that GoLiSA produces more precise results in detecting non-deterministic behaviors, outperforming existing static analyzers.

Then, we evaluate the quality of our results on two specific real-world applications, to show how the static analyses discussed in Section 4 work and how the information is propagated in smart contracts. In particular, we selected the first application from the HF benchmark, while the second one is a Cosmos SDK application.¹⁸

All the experiments was performed on a HP EliteBook 850 G4 equipped with an Intel Core i7-7500U at 2,70/2,90 GHz and 16 GB of RAM running Windows 10 Pro 64bit, Oracle JDK version 13, and Go version 1.17.

5.1 Quantitative Evaluation

The experimental artifact set has been retrieved from 954 GitHub repositories, by querying for the `chaincode` keyword, as smart contracts are called in HF, and selecting chaincodes from unforked Go repositories only¹⁹, that include the `Invoke` and `Init` methods: these are the transaction requests' entry points for chaincodes.²⁰ Then, we filtered out files unrelated to smart contracts and removed chaincodes not analyzable due of failures either GoLiSA or the tools discussed in Sect. 5.1.1. In particular, GoLiSA failures on such chaincodes are due to current missing support of high-order functions, recursion, and C code invocation via the built-in Go `cmd/go` package.²¹ This resulted in a benchmark consists of 651 chaincodes only (~ 167391 LoCs), that, from here on, we refer to as `HF`. Then, each chaincode has been manually inspected before applying GoLiSA to search for critical non-deterministic behavior. In particular, for each chaincode, we manually searched for sources of non-determinism (if present) and checked if the result of the corresponding instructions could have an impact (i.e., an update) on the blockchain global state or on the response. If so, we classified this behaviour as critical/harmful. On the selected benchmark, we have found a total of 124 critical/harmful non-deterministic behaviours. In our evaluation, a warning raised by an

¹⁷ An industrial application of GoLiSA for detecting non-determinism in Cosmos SDK can be found here [36].

¹⁸ The example reported in Figure 1 contains a snippet of code of this application

¹⁹ <https://api.github.com/search/repositories?q=chaincode+fork:false+language:Go+archived:false&sort=stars&order=desc>. Accessed: 17-10-2022.

²⁰ See <https://pkg.go.dev/github.com/hyperledger/fabric-chaincode-go/shim>.

²¹ We decided not to implement those standard features since this would have required a relevant effort to support only a few more chaincodes.

■ **Table 3** Analysis evaluation.

Analysis	#A	#U	ET	AT	#W	#TP	#FP	#FN
<i>Taint</i>	68	583	2h:15m:03s	12.45s	173	118	55	7
<i>Non-interference</i>	69	582	2h:25m:18s	13.39s	195	124	71	0

analyzer has been classified as true positive (TP) if it was part of the 124 critical behaviours mentioned above, and as false positive (FP) if not. All the critical behaviours, part of the 124 manually detected, for which there was no warning, have been marked as false negative (FN).

Table 3 reports the results of the experimental evaluation of GoLiSA over the benchmark $\mathbb{H}\mathbb{F}$, where **#A** is the number of affected chaincodes (i.e., chaincodes where at least a warning was issued), **#U** is the number of unaffected chaincodes (i.e., chaincodes where no warning was raised), **ET** is the total execution time, **AT** is the average execution time, **#W** is the total number of warnings issued, **#TP** is the number of true positives among the raised warnings, **#FP** is the number of false positives among the raised warnings, and **#FN** is the number of false negatives. In terms of execution time, the analyses performed averagely in around 15 seconds per chaincode. The experiments shows that *Non-interference* performs better than *Taint* in terms of precision, being able to detect all the true positives contained in $\mathbb{H}\mathbb{F}$, with a ratio of false positives less than 40%. This was expected since, as we have already discussed in Section 4 and unlike *Non-interference*, *Taint* is only able to track explicit information flows. In fact, the 7 false negatives (column **#FN** of Table 3) produced by *Taint* correspond to implicit non-deterministic behaviors.

5.1.1 Comparison

We compared GoLiSA with the open-source static analyzers for Go chaincode described in Section 6, namely ChainCode Analyzer and Revive^{CC}. Table 4 reports the comparison between GoLiSA and these tools over the same benchmark $\mathbb{H}\mathbb{F}$ discussed in Section 5.1.

The comparison shows that GoLiSA - *Non-interference* finds all the true issues contained in the benchmark, achieving the best and most accurate result in terms of precision with a 36.41% false positives ratio. Instead, although it has some false negatives, GoLiSA - *Taint* is the analysis with the lowest percentage of false positives with the 31.79% .

Revive^{CC} triggers 351 warnings out of which 77.49% are false positives. The only non-deterministic behaviour not detected by Revive^{CC} (last column) is due to the fact that it considers the `ioutil.ReadFile` API as safe, although reading a file should be considered non-deterministic in the blockchain context. Finally, ChainCode Analyzer is more precise w.r.t. Revive^{CC}, with 66.50% of false positives, but it has also the greatest number of false negatives, failing to detect a huge number of critical non-deterministic behaviors. This can be attributed to the fact that ChainCode Analyzer does not consider several APIs leading to non-determinism as critical and it fails to soundly detect iteration over maps.

Note that the amount of true positives discovered by GoLiSA analyses differs from the ones of other tools. In fact, GoLiSA is the only tool involved in our comparison that issues warnings on sinks rather than sources. This translates to fewer alarms being issued whenever values of multiple sources flow to the same sink (here, GoLiSA issues a single warning, while other tools issue one for each source), and to more alarms being raised whenever the value of a single source flows to multiple sinks (here instead, other tools issue a single warning, while GoLiSA issues one for each sink).

■ **Table 4** Warnings triggered by the analyzers on $\mathbb{H}\mathbb{F}$.

Tools	# W	# TP	# FP	# FN
GoLiSA - <i>Taint</i>	173	118	55	7
GoLiSA - <i>Non-interference</i>	195	124	71	0
ChainCode Analyzer	203	68	135	53
Revive [^] CC	351	79	272	1

```

1 func (s *SmartContract) registrarBoleto(APIStub shim.ChaincodeStubInterface, args []
   string) sc.Response {
2 // [...]
3 objBoleto.CodigoBarra = strconv.Itoa((rand.Intn(5) + 10000000 + // [...]
4 var notExpiredDate = time.Now()
5 objBoleto.DataVencimento = notExpiredDate.Format("02/01/2006")
6 // [...]
7 boletoAsBytes, _ := json.Marshal(objBoleto)
8 APIStub.PutState(args[0], boletoAsBytes)
9 // [...]
10 }

```

■ **Figure 10** Method `registrarBoleto` of `boleto` contract.

5.2 Qualitative Evaluation

5.2.1 Explicit Flow: the Boleto Contract

The `boleto` contract²², taken from $\mathbb{H}\mathbb{F}$, comes with a real non-determinism issue that can be found with explicit flows, and that was also detected by other tools during the comparison of Section 5.1.1. The `boleto` contract (Figure 10) seems to be a proof of concept application handling tickets in an e-commerce store, with the method `registrarBoleto` used to register a ticket.

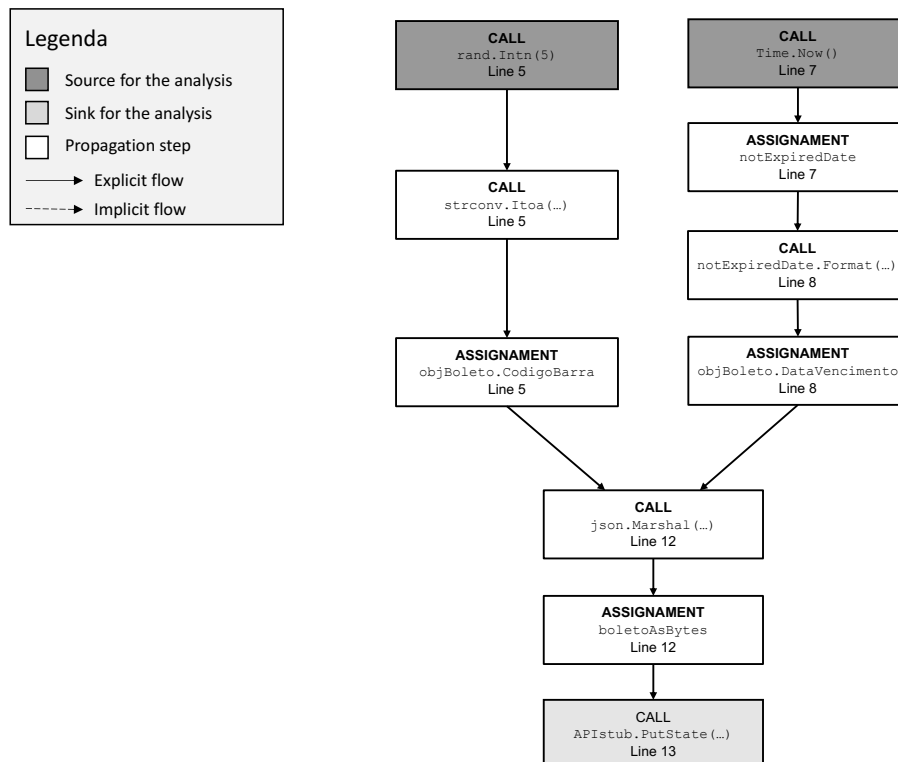
Analyzing `boleto`, GoLiSA detects the explicit flow leading to a non-deterministic behavior with both *Taint* and *Non-interference*. Method `registrarBoleto` contains two different sources of non-determinism that directly flow into the same sink. The first source detected by GoLiSA is the usage of the *Random API* to generate a barcode at line 3. Instead, the second source is the usage of the *OS API* that retrieves the local machine's time to set a date at line 4. As values from both sources are used to update fields of `objBoleto`, the latter is marked as tainted by the analysis, resulting in `boletoAsBytes` being tainted as well. As reported in Table 2, `PutState`'s parameters are considered as sinks by GoLiSA's analyses. According to the official documentation of $\mathbb{H}\mathbb{F}$ ²³, the `PutState` method does not affect the ledger until the transaction is validated and successfully committed. However, a transaction needs to produce the same results among different peers to be validated. Hence, as passing non-deterministic values to `PutState` will cause the transaction to fail, GoLiSA raises a warning on line 8.

5.2.2 Implicit Flow: Cosmos SDK v.43

Analyzing the code in Figure 1, GoLiSA is able to detect an implicit flow that leads to a non-deterministic behavior, that can only be detected using *Non-interference*. The `ValidateBasic` method of Cosmos SDK v. 0.43.x and v. 0.44. $\{0,1\}$ was designed to validate

²² <https://github.com/arthurmsouza/boleto/blob/master/boleto-chaincode/boleto.go>

²³ <https://github.com/hyperledger/fabric-chaincode-go/blob/1476cf1d3206f620db7eea12312c98669d39fa22/shim/interfaces.go>.



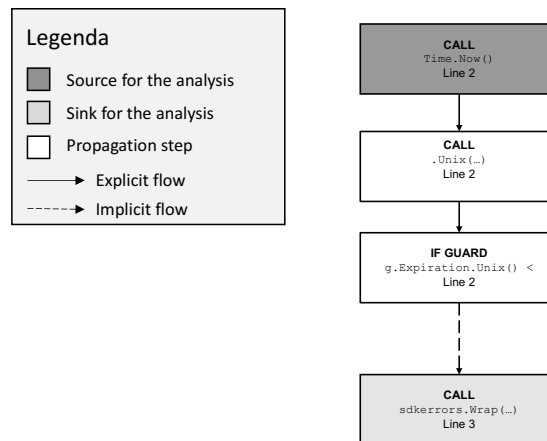
■ **Figure 11** Simplified view of explicit flow computed by GoLiSA during the analysis of registrarBoleto.

a grant to ensure it has not yet expired. In this case, the source detected by GoLiSA is the *OS API* used to retrieve the local machine time involved in the expiration check of the grant time at line 2 of Figure 1. By propagating the information, GoLiSA detects that the expiration check governs the execution of return statement. Since the *Wrap* method is annotated as a sink, GoLiSA triggers an alarm at line 3 of Figure 1 as the sink is contained in a block whose guard depends on non-deterministic values.

5.3 Limits

Unlike some frameworks and GPLs used in other blockchains, frameworks targeted by this paper are used to develop *permissioned*, and often *private*, blockchains, meaning that the related software is not publicly available or released with open-source licenses. This is also the reason why the benchmark $\mathbb{H}\mathbb{F}$ crawled from GitHub consists of 651 chaincodes, a number that is not comparable with smart contract benchmarks obtained investigating other (public and permissioned) blockchains. For instance, [44] collects 3075 distinct smart contracts from the Ethereum blockchain, resulting in a wider benchmark.

The proposed solution for detecting non-deterministic behaviors is fully static. It is well known that static analysis is intrinsically conservative and may produce false positives. Even if few have been raised by GoLiSA on the selected benchmark, one should expect more false positives when applying our approach to arbitrary DApps.



■ **Figure 12** Simplified view of implicit flow computed by GoLiSA during the analysis of Figure 1.

6 Related Work

The non-determinism of smart contracts written in GPLs is a well-known issue [32, 45]. Frameworks such as Takamaka [41, 42] enforce determinism adopting a conservative approach that limits the set of instructions and APIs of the target language, avoiding unsafe statements that might lead to non-deterministic behaviors through white-listing fully deterministic APIs. This approach ensures safe development while preventing that API extensions coming with new language versions can bypass the check. However, it also severely limits the exploitable features of the GPL. On the other hand, black-listing undesired APIs is a much harder approach to maintain, but it seems the most widespread technique in Go analyzers. For instance, ChainCode Analyzer [31] and Revive^{CC} [40] detect mainly black-listed imports related non-deterministic APIs using a syntactical approach. Besides, they can detect non-deterministic map iterations by AST traversal with minimal syntactic reasoning. Signature of invoked functions can also be black-listed instead of imports [32]. These tools and frameworks inherently limit API usage, sensibly reducing the benefits of adopting a GPL even when the code poses no harm to the blockchain. The problem of detecting non-determinism has also been covered for parallel applications, suggesting that non-determinism is “*most often the result of a mistake on the part of the programmer*” [13].

7 Conclusion

In this paper, we proposed a flow-based approach for detecting critical non-deterministic behaviors, namely the ones affecting the blockchain state. Our proposal has been implemented in GoLiSA, a static analyzer for Go applications. To the best of our knowledge, GoLiSA is the first semantic-based static analyzer for blockchain software able to detect non-deterministic behaviors, with an extremely low false alarm prevision. In the context of smart contracts, the proposed approach is placed in an off-chain architecture, i.e., the analysis is done before smart contracts are deployed in the blockchain, and it is not mandatory. As future work, besides supporting the missing Go features discussed in Section 5 to enhance the analysis coverage, we plan to test GoLiSA in an on-chain architecture [35], making the non-determinism checker part of the consensus protocol, with the goal of keeping the code stored within the blockchain deterministic. The analysis could be enriched with a context-sensitive flow reconstructor,

such as BackFlow [21], that starting from the results of a information flow engine, reconstructs how the information flows inside the program and builds paths connecting sources to sinks. Moreover, we have focused on the non-determinism problem, but our future research will address the problem of detecting other and equally critical vulnerabilities that can affect blockchain software written using general-purpose languages, such as numerical overflow.

Our proposal follows a fully static approach, justified by the fact that we aim at proving the determinism of blockchain software, regardless of the possible executions. However, even if the evaluation on the selected benchmark shows optimal results, the risk of getting false alarms analyzing other applications is still present, being our approach based on over-approximating possible executions via abstract interpretation. In future works, hybrid approaches between static and dynamic analyses will be investigated to get the benefits of both techniques.

Finally, in order to assess the effectiveness of our proposal, we have conducted our evaluation on Hyperledger Fabric blockchain software, mostly because it is the most popular framework among those cited in the paper. To give a larger coverage to GoLiSA of the blockchain software that can analyze, the next step will be to design significant benchmarks also for the other frameworks, such as Tendermint core and Cosmos SDK, on which we can experiment our static analyzer.

References

- 1 Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language, 1994. Accessed: 01-12-2022. URL: <https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf>.
- 2 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 30:1–30:15. ACM, 2018. doi:10.1145/3190508.3190538.
- 3 A. M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. O’Reilly, 2nd edition, 2017.
- 4 A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. O’Reilly, 2018.
- 5 Sotirios Brotsis, Nicholas Kolokotronis, Konstantinos Limniotis, Gueltoum Bendiab, and Stavros Shiaeles. On the security and privacy of hyperledger fabric: Challenges and open issues. In *2020 IEEE World Congress on Services (SERVICES)*, pages 197–204, 2020. doi:10.1109/SERVICES48979.2020.00049.
- 6 E. Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. PhD thesis, University of Guelph, 2016.
- 7 Ethan Buchman. Byzantine Fault Tolerant State Machine Replication in Any Programming Language. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC ’19*, page 546, New York, NY, USA, 2019. Association for Computing Machinery.
- 8 V. Buterin. Ethereum Whitepaper, 2013. Available at <https://ethereum.org/en/whitepaper/>.
- 9 Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Arash Pourdamghani. Probabilistic Smart Contracts: Secure Randomness on the Blockchain. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019*, pages 403–412. IEEE, 2019. doi:10.1109/BLOC.2019.8751326.
- 10 Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.
- 11 Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976. doi:10.1145/360051.360056.

- 12 ebuchman. Cosmos-SDK Vulnerability Retrospective: Security Advisory Jackfruit, October 12, 2021, 2021. Accessed: 01-12-2022. URL: <https://forum.cosmos.network/t/cosmos-sdk-vulnerability-retrospective-security-advisory-jackfruit-october-12-2021/5349>.
- 13 Perry A. Emrath and David A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, pages 89–99, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/68210.69224.
- 14 Michael D. Ernst, Alberto Lovato, Damiano Macedonio, Ciprian Spiridon, and Fausto Spoto. Boolean Formulas for the Static Identification of Injection Attacks in java. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2015. doi:10.1007/978-3-662-48899-7_10.
- 15 Pietro Ferrara. A generic framework for heap and value analyses of object-oriented programming languages. *Theor. Comput. Sci.*, 631:43–72, 2016. doi:10.1016/j.tcs.2016.04.001.
- 16 Pietro Ferrara, Elisa Burato, and Fausto Spoto. Security Analysis of the OWASP Benchmark with Julia. In *Proceedings of the First Italian Conference on Cybersecurity (ITASEC17), Venice, Italy, January 17-20, 2017*, volume 1816 of *CEUR Workshop Proceedings*, pages 242–247. CEUR-WS.org, 2017. Accessed: 01-12-2022. URL: <http://ceur-ws.org/Vol-1816/paper-24.pdf>.
- 17 Pietro Ferrara, Amit Kr Mandal, Agostino Cortesi, and Fausto Spoto. Static analysis for discovering IoT vulnerabilities. *Int. J. Softw. Tools Technol. Transf.*, 23(1):71–88, 2021. doi:10.1007/s10009-020-00592-x.
- 18 Pietro Ferrara and Luca Negrini. SARL: Oo framework specification for static analysis. In Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel, editors, *Software Verification*, pages 3–20, Cham, 2020. Springer International Publishing.
- 19 Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. Static analysis for dummies: experiencing lisa. In Lisa Nguyen Quang Do and Caterina Urban, editors, *SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021*, pages 1–6. ACM, 2021. doi:10.1145/3460946.3464316.
- 20 Pietro Ferrara, Luca Olivieri, and Fausto Spoto. Tailoring Taint Analysis to GDPR. In *Privacy Technologies and Policy - 6th Annual Privacy Forum, APF 2018, Barcelona, Spain, June 13-14, 2018, Revised Selected Papers*, volume 11079 of *Lecture Notes in Computer Science*, pages 63–76. Springer, 2018. doi:10.1007/978-3-030-02547-2_4.
- 21 Pietro Ferrara, Luca Olivieri, and Fausto Spoto. Backflow: Backward context-sensitive flow reconstruction of taint analysis results. In *Verification, Model Checking, and Abstract Interpretation*, pages 23–43, Cham, 2020. Springer International Publishing.
- 22 Pietro Ferrara, Luca Olivieri, and Fausto Spoto. Static Privacy Analysis by Flow Reconstruction of Tainted data. *Int. J. Softw. Eng. Knowl. Eng.*, 31(7):973–1016, 2021. doi:10.1142/S0218194021500303.
- 23 Luca Foschini, Andrea Gavagna, Giuseppe Martuscelli, and Rebecca Montanari. Hyperledger Fabric Blockchain: Chaincode Performance Analysis. In *2020 IEEE International Conference on Communications, ICC 2020, Dublin, Ireland, June 7-11, 2020*, pages 1–6. IEEE, 2020. doi:10.1109/ICC40277.2020.9149080.
- 24 Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.
- 25 Joseph A. Goguen and José Meseguer. Unwinding and Inference Control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 29 - May 2, 1984*, pages 75–87. IEEE Computer Society, 1984. doi:10.1109/SP.1984.10019.


- 26 Hyperledger. Hyperledger fabric documentation. URL: <https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html#what-is-hyperledger-fabric>.
- 27 Tendermint Inc. What is Tendermint: A Note on Determinism, 2022. Accessed: 01-12-2022. URL: <https://github.com/tendermint/tendermint/blob/7983f9cc36c31e140e46ae5cb00ed39f637ef283/docs/introduction/what-is-tendermint.md#a-note-on-determinism>.
- 28 Uday P. Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Compiler Construction*, pages 213–228, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78791-4_15.
- 29 J. Kwon. Tendermint: Consensus without mining, 2014.
- 30 J. Kwon and E. Buchman. Cosmos whitepaper, 2019.
- 31 kzhry. Chaincode Analyzer, 2021. Accessed: 01-12-2022. URL: <https://github.com/hyperledger-labs/chaincode-analyzer>.
- 32 Penghui Lv, Yu Wang, Yazhe Wang, and Qihui Zhou. Potential Risk Detection System of Hyperledger Fabric Smart Contract based on Static Analysis. In *IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021*, pages 1–7. IEEE, 2021. doi:10.1109/ISCC53001.2021.9631249.
- 33 S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Available at <https://bitcoin.org/bitcoin.pdf>, 2008.
- 34 Luca Negrini. *A generic framework for multilanguage analysis*. PhD thesis, Università Ca' Foscari Venezia, 2023.
- 35 Luca Olivieri, Fausto Spoto, and Fabio Tagliaferro. On-Chain Smart Contract Verification over Tendermint. In *Financial Cryptography and Data Security. FC 2021 International Workshops - CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers*, volume 12676 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2021. doi:10.1007/978-3-662-63958-0_28.
- 36 Luca Olivieri, Fabio Tagliaferro, Vincenzo Arceri, Marco Ruaro, Luca Negrini, Agostino Cortesi, Pietro Ferrara, Fausto Spoto, and Enrico Talin. Ensuring determinism in blockchain software with golisa: an industrial experience report. In Laure Gonnord and Laura Titolo, editors, *SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*, pages 23–29. ACM, 2022. doi:10.1145/3520313.3534658.
- 37 Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.
- 38 A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. doi:10.1109/JSAC.2002.806121.
- 39 Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences, 1978.
- 40 sivachokkapu. Revivecc, 2021. Accessed: 01-12-2022. URL: <https://github.com/sivachokkapu/revive-cc>.
- 41 Fausto Spoto. A Java Framework for Smart Contracts. In *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*, volume 11599 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2019. doi:10.1007/978-3-030-43725-1_10.
- 42 Fausto Spoto. Enforcing Determinism of Java Smart Contracts. In *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, volume 12063 of *Lecture Notes in Computer Science*, pages 568–583. Springer, 2020. doi:10.1007/978-3-030-54455-3_40.
- 43 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009. doi:10.1145/1542476.1542486.

- 44 Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA):189:1–189:29, 2019. doi:10.1145/3360615.
- 45 Kazuhiro Yamashita, Yoshihide Nomura, Ence Zhou, Bingfeng Pi, and Sun Jun. Potential Risks of Hyperledger Fabric Smart Contracts. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 1–10, 2019. doi:10.1109/IWBOSE.2019.8666486.

Toward Tool-Independent Summaries for Symbolic Execution

Frederico Ramos ✉ 🏠 

Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Nuno Sabino ✉ 🏠 


Instituto Superior Técnico, University of Lisbon, Portugal
Carnegie Mellon University, Pittsburgh, PA, USA
Institute of Telecommunications, Campus de Santiago, Aveiro, Portugal

Pedro Adão ✉ 🏠 

Instituto Superior Técnico, University of Lisbon, Portugal
Institute of Telecommunications, Campus de Santiago, Aveiro, Portugal

David A. Naumann ✉ 🏠 

Stevens Institute of Technology, Hoboken, NJ, USA

José Fragoso Santos ✉ 🏠 

Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Abstract

We introduce a new symbolic reflection API for implementing tool-independent summaries for the symbolic execution of C programs. We formalise the proposed API as a symbolic semantics and extend two state-of-the-art symbolic execution tools with support for it. Using the proposed API, we implement 67 tool-independent symbolic summaries for a total of 26 LIBC functions. Furthermore, we present SUMBOUNDVERIFY, a fully automatic summary validation tool for checking the bounded correctness of the symbolic summaries written using our symbolic reflection API. We use SUMBOUNDVERIFY to validate 37 symbolic summaries taken from 3 state-of-the-art symbolic execution tools, *angr*, *Binsec* and *Manticore*, detecting a total of 24 buggy summaries.

2012 ACM Subject Classification Software and its engineering → Software verification and validation; Security and privacy → Formal methods and theory of security

Keywords and phrases Symbolic Execution, Runtime Modelling, Symbolic Summaries

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.24

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.9.2.7>

Funding The authors were supported by Fundação para a Ciência e a Tecnologia (UIDB/50008/2020, Instituto de Telecomunicações, and UIDB/50021/2020, INESC-ID multi-annual funding, and PhD grant SFRH/BD/150692/2020), project DIVINA (CMU/TIC/0053/2021), the SmartRetail project (C6632206063-00466847) financed by IAPMEI, the European Commission under grant agreement number 830892 (SPARTA), and the NSF award CNS-1718713.

1 Introduction

Symbolic execution [14, 34] is a program analysis technique that allows for the exploration of all the execution paths of the given program up to a bound, by executing the program with symbolic values instead of concrete ones. For each execution path, the symbolic execution engine builds a first order formula, called path condition, which accumulates the constraints on the symbolic inputs that cause the execution to follow that path. Symbolic execution engines rely on an underlying SMT solver [20, 9] to check the feasibility of execution paths



© Frederico Ramos, Nuno Sabino, Pedro Adão, David A. Naumann, and José Fragoso Santos;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 24; pp. 24:1–24:29



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



and the validity of any assertions supplied by the developer. Despite being extensively used in practice [15, 26, 7, 54], symbolic execution suffers from two main limitations when applied to real-world code: interactions with the runtime environment (e.g. file system, network, operating system) and path explosion. An effective approach to deal with these two issues is to use symbolic summaries to model the behaviour of both external runtime functions as well as internal functions with a high level of branching [6].

Symbolic summaries constrain the symbolic state of the given program so as to simulate the behaviour of the modelled functions without having to symbolically execute them. The idea is that instead of symbolically executing the code of a given concrete function on some symbolic inputs, one implements a symbolic summary that models the behaviour of that function, and then executes the summary instead of the concrete function. Importantly, symbolic summaries allow developers to merge different symbolic execution paths into a single path by explicitly interacting with the current symbolic state [6, 52]. Hence, they provide an effective mechanism for containing the number of paths to be explored during symbolic execution, allowing developers to mitigate the effect of the path explosion problem.

When writing a symbolic summary, tool developers must carefully construct the symbolic state that properly captures the outcome of all the execution paths that the summary is supposed to model. They do this by directly interacting with the various elements of the given symbolic state using symbolic reflection mechanisms [6, 52]. This is often a challenging task that is both error prone and difficult to validate. For this reason, most symbolic execution tools for C have very limited support for external functions and commonly used library functions, such as those of the Standard C Library (libc).

State-of-the-art symbolic execution tools [50, 15, 38] come with their own symbolic summaries implemented in the programming languages used to build each tool. For instance, *angr*'s [50] summaries are implemented in Python, *KLEE*'s [15] summaries in C, and *BINSEC*'s [19] summaries in OCaml, even though all these tools target C code. These summaries often rely on specific aspects of the tools for which they were implemented, making it extremely difficult to share summaries between different symbolic execution tools. Surprisingly, and although there is a clear lack of appropriate tool support for developing and sharing symbolic summaries across different symbolic execution tools, the research community has not yet given much attention to this topic. The current state of affairs is, however, dire: even though the Standard C Library (libc) includes more than one thousand functions, the symbolic execution tool with the broadest support for libc is *angr* [50], with only 128 unverified symbolic summaries. This situation is made considerably worse by the fact that even the few existing summaries are written manually and not verified, potentially compromising the correctness and coverage guarantees of their corresponding symbolic execution tools.

In this paper, we introduce a new symbolic reflection API for the implementation of tool-independent symbolic summaries for the C programming language. The proposed API consists of a set of symbolic reflection primitives [52] for the explicit manipulation of C symbolic states in a tool-independent way. Our symbolic primitives include a variety of instructions for: creating symbolic variables and first-order constraints, checking the satisfiability of constraints, and extending the path condition of the current symbolic state with a given constraint. Symbolic summaries implemented using our API are written directly in C and can therefore be shared across different symbolic execution tools, provided that these tools implement the proposed API. Importantly, the goal of our API is not to make symbolic summaries simpler or easier to write, but rather to establish a symbolic reflection interface shared by all symbolic execution tools, allowing for the decoupling of symbolic summaries from

the internal details of each tool. To illustrate the applicability of our API, we have extended the symbolic execution tools *anqr* [50] and *AVD* [45] with support for it and developed 67 tool-independent symbolic summaries for a total of 27 LIBC functions, including string-manipulating, number-parsing, input/output, and heap-manipulating functions. Furthermore, we formalised our symbolic reflection API as a symbolic semantics [52, 21] and used this semantics to formally characterise the correctness properties that symbolic summaries are expected to observe, specifically, over- and/or under-approximation.

Leveraging our symbolic reflection API, we developed *SUMBOUNDVERIFY*, a new fully-automated tool for the bounded verification of symbolic summaries. *SUMBOUNDVERIFY* works by comparing the execution paths modelled by the given summary against those generated by symbolically executing its corresponding concrete function up to the chosen bound. In order to assess the effectiveness of *SUMBOUNDVERIFY*, we used it to verify summaries belonging to three state-of-the-art symbolic execution tools: *anqr* [50], *Binsec* [19, 39, 18], and *Manticore* [38]. Out of the 37 analysed summaries, 24 were flagged as buggy, clearly demonstrating the need for tool support when it comes to designing and implementing symbolic summaries. This need is further confirmed by our own experience in the development of symbolic summaries, which we typically found to be highly complex and error-prone. This paper bridges this gap by providing the first verification tool specifically aimed at the development of correct symbolic summaries.

In summary, the contributions of this paper are the following: (1) a formally defined API for developing symbolic summaries for C; (2) a library of 67 symbolic summaries modelling 26 LIBC functions; and (3) *SUMBOUNDVERIFY*, an automatic bounded verification tool for validating symbolic summaries against their corresponding concrete implementations.

2 Overview

In this section, we first contrast the existing methodology for implementing symbolic summaries with our proposed approach (§2.1) and then give a high-level overview of *SUMBOUNDVERIFY* (§2.2), illustrating how it can be used to verify symbolic summaries.

2.1 Tool-Specific vs. Tool-Independent Symbolic Summaries

A symbolic summary is an operational model of a function that simulates its behaviour by interacting directly with the underlying symbolic state. So far, each symbolic execution tool for C comes with its own summaries directly implemented in the programming language used to build the tool. Existing symbolic summaries are therefore tightly connected to the architecture of their corresponding tools, preventing summaries from being shared between different tools. In order to cater for the reuse of symbolic summaries, we propose an alternative approach: *Symbolic summaries are to be directly implemented in C using a shared symbolic reflection API for direct manipulation of symbolic states at the programming language level.*

To illustrate the difference between tool-specific and tool-independent summaries, we compare Manticore’s [38] symbolic summary for `strlen` (Figure 1) with the equivalent summary written directly in C using our API (Figure 2).

Manticore’s Tool-Specific Summary

Figure 1 shows Manticore’s summary for the function `strlen`. This summary first checks if the given string pointer is itself symbolic, in which case it throws an error (lines 3-4). Then, the summary uses the Manticore’s internal function `find_zero` to determine the index of the

24:4 Toward Tool-Independent Summaries for Symbolic Execution

```
1 def strlen_approx(state: State, s: Union[int, BitVec]) -> Union[int, BitVec]:
2
3     if issymbolic(s):
4         raise ConcretizeArgument(state.cpu, 1)
5
6     #Find max string length
7     cpu = state.cpu
8     zero_idx = _find_zero(cpu, state, s)
9     ret = zero_idx
10
11     #Build nested ITE formula
12     for offset in range(zero_idx - 1, -1, -1):
13         byt = cpu.read_int(s + offset, 8)
14         if issymbolic(byt):
15             ret = ITEBV(cpu.address_bit_size, byt == 0, offset, ret)
16
17     return ret
```

■ **Figure 1** Implementation of Manticore’s `strlen` summary.

first concrete null character in the input string, `zero_idx`. Note that if the string does not contain any symbolic character, `zero_idx` coincides with the length of the string. In the final for-loop, the summary iterates over the characters of the given string to construct a symbolic expression denoting its length. For instance, given the symbolic string `[c0, c1, c2, \0]`, the loop will generate the expression:

$$\text{ret} = \text{ITE}(c0 == \backslash 0, 0, \overline{\text{ITE}(c1 == \backslash 0, 1, \overline{\text{ITE}(c2 == \backslash 0, 2, 3))})})$$

signifying that: if the first character (`c0`) is null, then the return value is 0; if the second character (`c1`) is null, then the return value is 1; if the third character (`c2`) is null, then the return value is 2; otherwise, the return value is 3. Note that the overlines have no semantic meaning, being only there to facilitate the reading.

Tool-Independent Summary

Figure 2 shows our equivalent C implementation of Manticore’s summary for `strlen`. Although both summaries implement approximately the same logic,¹ our summary is written directly in C using our symbolic reflection API. It uses the following primitives: (1) `is_symbolic(x)` to check if variable `x` denotes a symbolic value; (2) `new_sym_var(size)` to create a new symbolic variable to represent a value of size `size`; (3) `_solver_EQ(a, b)` to build a constraint stating that the two given values are equal; (4) `assume(c)` to add the constraint `c` to the path condition of the current symbolic state; and (5) the primitive `_solver_IF(c, a, b)` to build an if-then-else symbolic expression of the form `ITE(c, a, b)`.

Why use symbolic summaries?

To better understand the benefits of symbolic summaries, let us consider the symbolic execution of the concrete implementation of `strlen` on the symbolic string `[c0, c1, c2, \0]`. That execution would generate four execution paths, each corresponding to one of the possible outputs. In contrast, the execution of either of the symbolic summaries described above generates a single execution path representing all four outputs. Both symbolic summaries

¹ The C summary assumes that it is never given a symbolic pointer as input.

```

1 size_t strlen(char* s){
2     int i = 0;
3     char char_zero = '\0';
4
5     //Calculate max string length
6     while(is_symbolic(&s[i]) || s[i] != '\0'){
7         i++;
8     }
9     int len = i;
10    symbolic ret = new_sym_var(INT_SIZE);
11    cnstr_t ret_cnstr = _solver_EQ(&ret, &len, INT_SIZE);
12
13    //Build nested ITE constraint
14    for(i = len-1; i >= 0; i--){
15
16        if(is_symbolic(&s[i])){
17
18            cnstr_t c_eq_zero = _solver_EQ(&s[i], &char_zero, CHAR_SIZE);
19            cnstr_t ret_eq_i = _solver_EQ(&ret, &i, INT_SIZE);
20
21            ret_cnstr = _solver_IF(c_eq_zero, ret_eq_i, ret_cnstr);
22        }
23    }
24    assume(ret_cnstr);
25    return ret;
26 }

```

■ **Figure 2** Implementation of *Manticore's* `strlen` in C.

achieve this by directly extending the path condition of the calling state with a formula that constrains the return value appropriately, depending on the symbolic characters appearing in the given string.

2.2 Bounded Verification of Symbolic Summaries

In this section we show how `SUMBOUNDVERIFY` can be used to verify the symbolic summary of `strlen` given above. We support two flavours of correctness properties: *under-approximating* [40] and *over-approximating* [5]. A summary is *under-approximating* if all the execution paths modelled by the summary are contained in the set of concrete paths of its corresponding function. In other words, an *under-approximating* summary guarantees that all its generated paths have corresponding concrete paths. Conversely, a symbolic summary is *over-approximating* if it models all the concrete paths of its corresponding function; that is, an over-approximating summary must take into account all possible concrete paths. If a summary is both under- and over-approximating, we say that it is *exact*, following recent terminology in the context of separation-logic-based verification [37]. Naturally, the type of correctness property to be aimed at depends on how the summary is going to be used. For instance, over-approximating summaries are essential for security applications that must guarantee the absence of security vulnerabilities; in contrast, under-approximating summaries may be a better fit for debugging tools that aim at reporting only real bugs.

Bounded Verification

Let us now take a closer look at the inner workings of `SUMBOUNDVERIFY`. In a nutshell, `SUMBOUNDVERIFY` requires the developer to provide the summary to be verified, its corresponding concrete implementation, and an integer bound on the size of its parameters; for instance, for inputs of array type, this bound corresponds to the maximum length of

```

1 int main(){
2
3 char s[4];
4 for (int i = 0; i < 2; i++){
5   s[i] = new_sym_var_array("c", i, CHAR_SIZE);
6 }
7 s[3] = '\0';
8
9 state_t fresh_state = save_current_state();
10
11 int ret1 = concrete_strlen(s);
12 cnstr_t c1 = get_cnstr(&ret1, INT_SIZE);
13 store_cnstr("cnstr", c1);
14
15 switch_state(fresh_state);
16
17 int ret2 = summ_strlen(s);
18 cnstr_t c2 = get_cnstr(&ret2, INT_SIZE);
19 store_cnstr("summ", c2);
20
21 result_t res = check_implications("summ", "cnstr");
22 print_counterexamples(res);
23 return 0;
24 }

```

(a) Test code.

Reference Implementation Formula:

$$\begin{aligned}
&(c_0 = \backslash 0) \wedge (ret = 0) \vee \\
&(c_0 \neq \backslash 0) \wedge (c_1 = \backslash 0) \wedge (ret = 1) \vee \\
&(c_0 \neq \backslash 0) \wedge (c_1 \neq \backslash 0) \wedge (c_2 = \backslash 0) \wedge (ret = 2) \vee \\
&(c_0 \neq \backslash 0) \wedge (c_1 \neq \backslash 0) \wedge (c_2 \neq \backslash 0) \wedge (c_3 = \backslash 0) \\
&\wedge (ret = 3)
\end{aligned}$$

Symbolic Summary Formula:

$$\begin{aligned}
&ITE(c_0 = \backslash 0, r = 0, \\
&\quad ITE(c_1 = \backslash 0, r = 1, \\
&\quad\quad ITE(c_2 = \backslash 0, r = 2, 3)) \wedge ret = r
\end{aligned}$$

(b) Generated formulas.

■ **Figure 3** Bounded Verification of the `strlen` summary given in Figure 2.

the array. Given the signature of the summarised function, `SUMBOUNDVERIFY` synthesises a set of symbolic tests to check the correctness of the given summary. These tests can be executed by any symbolic execution tool that implements our reflection API. If the summary passes all the generated tests, then it is correct up to the pre-established bound. If it does not, then `SUMBOUNDVERIFY` generates a concrete input that is not correctly modelled by the summary.

Figure 3 illustrates one of the tests generated for the `strlen` summary discussed in §2.1, assuming that the developer specified bound 3. The test first creates an array of size 4, initialises the first 3 characters to new symbolic characters, and sets the fourth element of the array to be the null character (lines 3-7). Then, the test uses the API function `save_current_state` to save the current symbolic state (line 9). Next, the test calls the concrete `strlen` function on the created symbolic string and stores the generated return values and final path conditions for future reference (lines 12-14). Then, the test re-establishes the symbolic state saved in line 10 by calling the API function `switch_state` (line 15), which simply replaces the current symbolic state with the given symbolic state. Having re-established the original symbolic state, the test calls the summary on the input string and stores the generated return values and final path conditions (lines 17-19). Finally, the test compares the return values and path conditions generated by the summary against those generated by the concrete function.

A summary can be classified as being: under-approximating correct, over-approximating correct, or incorrect. In a nutshell, the two correct cases are checked as follows:

- *Under-approximating*: the formula describing the final state resulting from the symbolic execution of the summary implies the formula describing the final state resulting from the symbolic execution of its reference implementation;
- *Over-approximating*: the formula describing the final state resulting from the symbolic execution of the reference implementation implies the formula describing the final state resulting from the symbolic execution of the summary.

The `strlen` summary given in Figure 2 is exact (i.e., both under- and over-approximating). Figure 3b shows the formulas generated by the execution of both the reference implementation and the summary. As the summary is exact, the solver can check both implications.

3 Symbolic Reflection API

We formally define the semantics of our API for developing symbolic summaries (§3.1) and use this semantics to characterise the correctness properties that symbolic summaries are expected to observe (§3.2). Then, we illustrate how our reflection API can be used to develop symbolic summaries for string-manipulating and number-parsing LIBC functions (§3.3).

3.1 Formal Semantics

We define the formal semantics of our Symbolic Reflection API on top of a core C-like language in the style of the core language used in [41], which we extend with our symbolic reflection primitives. Importantly, and in order not to clutter the formalism with unnecessary technical details, the formal model is a simplified version of our proposed API. The syntax of the language is given in the table below.

Syntax

e	$:=$	$n \mid x \mid \ominus(e) \mid \oplus(e_1, e_2) \mid \otimes(e_1, e_2, e_3)$
\hat{s}	$:=$	$x := e \mid \mathbf{skip} \mid \hat{s}_1; \hat{s}_2 \mid \mathbf{if}(e) \{ \hat{s}_1 \} \mathbf{else} \{ \hat{s}_2 \} \mid \mathbf{while}(e) \{ \hat{s} \} \mid \mathbf{return} e$ $\mid x := e_1[e_2] \mid e_1[e_2] := e_3 \mid x := \mathbf{new}(e) \mid rs$
rs	$:=$	$\mathbf{assert}(e) \mid \mathbf{assume}(e) \mid x := \mathbf{is_symbolic}(e) \mid x := \mathbf{symb}() \mid x := \mathbf{is_sat}(e)$ $\mid x := \mathbf{maximize}(e) \mid x := \mathbf{minimize}(e) \mid x := \mathbf{cur_pc}() \mid x := \mathbf{eval}(e)$ $\mid x := \mathbf{block_size}(e) \mid x := \mathbf{constr}_{uop}(e) \mid x := \mathbf{constr}_{bop}(e_1, e_2)$
bop	$:=$	$\mathbf{or} \mid \mathbf{and} \mid \mathbf{eq} \mid \mathbf{neq} \mid \mathbf{lt} \mid \mathbf{le}$
uop	$:=$	\mathbf{not}

Expressions $e \in Expr$ include integers n , program variables x , unary, binary, and ternary operators. Statements $\hat{s} \in Stmt$ include: (i) the typical imperative statements, i.e. variable assignment, skip, sequence, if, while, and return statements; (ii) statements for interaction with a linear memory, and (iii) symbolic reflection primitives $rs \in \mathcal{RS}$. In the following we use \hat{s} for statements that may include reflection primitives and s for statements that do not. Accordingly, we use \hat{s} for symbolic summaries and s for reference implementations.

The statements for memory interaction are the following: (1) the statement $x := e_1[e_2]$ assigns to x the value stored in the memory block denoted by e_1 at the offset denoted by e_2 ; (2) the statement $e_1[e_2] := e_3$ stores the value denoted by e_3 in the memory block denoted by e_1 at the offset denoted by e_2 ; and (3) the statement $x := \mathbf{new}(e)$ creates a memory block with the size denoted by e and assigns the obtained pointer to x .

The symbolic reflection primitives $rs \in \mathcal{RS}$ are the following: (1) $\mathbf{assert}(e)$ to check if the current path condition implies the constraint denoted by e ; (2) $\mathbf{assume}(e)$ to extend the current path condition with the constraint denoted by e ; (3) $x := \mathbf{is_symbolic}(e)$ to assign to variable x a boolean value indicating if e denotes a symbolic expression; (4) $x := \mathbf{symb}()$ to assign a fresh symbolic value to x ; (5) $x := \mathbf{is_sat}(e)$ to check if the constraint denoted by e is satisfiable when conjoined with the current path condition; (6) $x := \mathbf{maximize}(e)$ to assign the largest possible value that may be denoted by e to x ; (7) $x := \mathbf{minimize}(e)$ to assign the smallest possible value that may be denoted by e to x ; (8) $x := \mathbf{cur_pc}()$ to assign the formula denoting the current path condition to x ; (9) $x := \mathbf{eval}(e)$ to assign one of the concrete values denoted by e to x ; (10) $x := \mathbf{block_size}(e)$ to assign the size of the memory block pointed to by e to x ; (11) $x := \mathbf{constr}_{uop}(e)$ to assign the constraint resulting from the application of the logical unary operator uop to the symbolic value denoted by e to x ; and (12) $x := \mathbf{constr}_{bop}(e_1, e_2)$ to assign the constraint resulting from the application of the logical binary operator bop to the symbolic values denoted by e_1 and e_2 .

$$\begin{array}{c}
\text{NEW} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = n \quad \text{alloc}(\hat{\mu}, n) = \langle \hat{\mu}', l \rangle \quad \hat{\rho}' = \hat{\rho}[x \mapsto l]}{\hat{\mu}, \hat{\rho}, x := \text{new}(e) \rightsquigarrow \hat{\mu}', \hat{\rho}', \mathbb{C}(\cdot)} \\
\\
\text{STORE} \\
\frac{\llbracket e_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket e_2 \rrbracket_{\hat{\rho}} = \hat{\delta} \quad \llbracket e_3 \rrbracket_{\hat{\rho}} = \hat{v} \quad \text{store}(\hat{\mu}, l, \hat{\delta}, \hat{v}) \rightsquigarrow \mathbb{S}(\hat{\mu}', \pi')}{\hat{\mu}, \hat{\rho}, \pi, e_1[e_2] := e_3 \rightsquigarrow \hat{\mu}', \hat{\rho}, \pi \wedge \pi', \mathbb{C}(\cdot)} \\
\\
\text{IF-TRUE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \hat{v} \quad \pi' = \pi \wedge (\hat{v} = \text{true})}{\hat{\rho}, \pi, \text{if}(e) \{ \hat{s}_1 \} \text{ else } \{ \hat{s}_2 \} \rightsquigarrow \hat{\rho}, \pi', \mathbb{C}(\hat{s}_1)} \\
\\
\text{IF-FALSE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \hat{v} \quad \pi' = \pi \wedge (\hat{v} = \text{false})}{\hat{\rho}, \pi, \text{if}(e) \{ \hat{s}_1 \} \text{ else } \{ \hat{s}_2 \} \rightsquigarrow \hat{\rho}, \pi', \mathbb{C}(\hat{s}_2)} \\
\\
\text{ASSIGNMENT} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \hat{v}}{\hat{\rho}, x := e \rightsquigarrow \hat{\rho}[x \mapsto \hat{v}], \mathbb{C}(\cdot)} \\
\\
\text{SEQUENCE - EMPTY} \\
\frac{\hat{\mu}, \hat{\rho}, \pi, \hat{s}_1 \rightsquigarrow \hat{\mu}', \hat{\rho}', \pi', \mathbb{C}(\cdot)}{\hat{\mu}, \hat{\rho}, \pi, \hat{s}_1; \hat{s}_2 \rightsquigarrow \hat{\mu}', \hat{\rho}', \pi', \mathbb{C}(\hat{s}_2)} \\
\\
\text{SEQUENCE - CONT} \\
\frac{\hat{\mu}, \hat{\rho}, \pi, \hat{s}_1 \rightsquigarrow \hat{\mu}', \hat{\rho}', \pi', \mathbb{C}(\hat{s}_1)}{\hat{\mu}, \hat{\rho}, \pi, \hat{s}_1; \hat{s}_2 \rightsquigarrow \hat{\mu}', \hat{\rho}', \pi', \mathbb{C}(\hat{s}_1; \hat{s}_2)} \\
\\
\text{LOAD} \\
\frac{\llbracket e_1 \rrbracket_{\hat{\rho}} = l \quad \llbracket e_2 \rrbracket_{\hat{\rho}} = \hat{\delta} \quad \text{load}(\hat{\mu}, l, \hat{\delta}) \rightsquigarrow_s \mathbb{S}(\hat{v}, \pi') \quad \hat{\rho}' = \hat{\rho}[x \mapsto \hat{v}] \quad \pi'' = \pi \wedge \pi'}{\hat{\mu}, \hat{\rho}, \pi, x := e_1[e_2] \rightsquigarrow \hat{\mu}, \hat{\rho}', \pi'', \mathbb{C}(\cdot)} \\
\\
\text{SKIP} \\
\text{skip} \rightsquigarrow \mathbb{C}(\cdot) \\
\\
\text{RETURN} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \hat{v}}{\hat{\rho}, \text{return } e \rightsquigarrow \hat{\rho}, \mathbb{R}(\hat{v})} \\
\\
\text{WHILE} \\
\text{while}(e) \{ \hat{s} \} \rightsquigarrow \mathbb{C}(\text{if}(e) \{ \hat{s}, \text{while}(e) \{ \hat{s} \} \} \text{ else } \{ \text{skip} \})
\end{array}$$

■ **Figure 4** Core Semantics: Imperative Fragment.

Symbolic Execution – Trace Semantics

The symbolic semantics of our core language operates on symbolic states, which store symbolic values given by the grammar: $\hat{v} \in \hat{\mathcal{V}} ::= n \mid \hat{x} \mid \ominus(\hat{v}) \mid \oplus(\hat{v}, \hat{v}) \mid \otimes(\hat{v}, \hat{v}, \hat{v})$. Symbolic values include: integers n , symbolic variables $\hat{x} \in \hat{\mathcal{X}}$, and unary, binary, and ternary operators, respectively ranged by \ominus , \oplus , and \otimes . Furthermore, we use $\pi \in \Pi$ to range over symbolic values of type Boolean. Symbolic states $\hat{\sigma} \in \text{SymSt}$ are composed of:

- a *symbolic heap*, $\hat{\mu} : \mathbb{N} \rightarrow \hat{\mathcal{V}} \times \mathbb{N} \times \mathbb{N}$, mapping integer pointers $l \in \mathbb{N}$ to triples of the form (\hat{v}, k_l, k_r) , where \hat{v} denotes the symbolic value stored at location l and k_l and k_r respectively denote the number of cells that can be accessed to the left and to the right of l using l as the accessing pointer;
- a *symbolic store*, $\hat{\rho} : \mathcal{X} \rightarrow \hat{\mathcal{V}}$, mapping program variables to symbolic values; and
- a *path condition*, $\pi \in \Pi$, keeping track of the constraints on which the current symbolic execution branched so far.

Note that our symbolic execution model requires heap locations to always be concrete; hence, we take the domain of symbolic heaps to be the set of naturals, \mathbb{N} , rather than that of symbolic values, $\hat{\mathcal{V}}$. Symbolic heaps, as concrete heaps, are organised in *blocks*, with each block being a sequence of memory locations; blocks can be univocally identified by their first location, which is referred to as the *head of the block* (i.e., l is the head of a block in $\hat{\mu}$, if $\hat{\mu}(l) = (-, 0, -)$, meaning that one cannot access any location before l within its block). Furthermore, we assume heaps to be *well-formed*, meaning that ranges of adjacent locations are consistent with each other; put formally, a heap $\hat{\mu}$ is said to be *well-formed* if and only if:

$$\begin{aligned}
\forall l \in \text{dom}(\hat{\mu}). \hat{\mu}(l) = (-, 0, k_r) &\implies \forall 0 \leq i < k_r. \hat{\mu}(l+i) = (-, i, k_r-i) \\
\forall l \in \text{dom}(\hat{\mu}). \hat{\mu}(l) = (-, k_l, k_r) &\implies \hat{\mu}(l-k_l) = (-, 0, k_l+k_r)
\end{aligned}$$

$$\begin{array}{c}
\text{ALLOC} \\
\frac{l = |\text{dom}(\hat{\mu})| \quad \hat{\mu}' = \hat{\mu}[l+i \mapsto (0, n-i, i) \mid 0 \leq i < n]}{\mathbf{alloc}(\hat{\mu}, n) \triangleq \langle \hat{\mu}', l \rangle} \\
\\
\text{BLOCK-SIZE} \\
\frac{\hat{\mu}(l) = (-, -, k_r)}{\mathbf{b_size}(\hat{\mu}, l) \triangleq k_r} \\
\\
\text{LOAD - IN BOUNDS} \\
\frac{\hat{\mu}(l) = (-, k_l, k_r) \quad -k_l \leq i < k_r \quad \pi' = \hat{\sigma} = i \quad \hat{\mu}(l+i) = (\hat{v}, -, -)}{\mathbf{load}(\hat{\mu}, l, \hat{\sigma}) \rightsquigarrow_s \mathbb{S}\langle \hat{v}, \pi' \rangle} \\
\\
\text{STORE - IN BOUNDS} \\
\frac{\hat{\mu}(l) = (-, k_l, k_r) \quad -k_l \leq i < k_r \quad \hat{\mu}(l+i) = (-, k'_l, k'_r) \quad \hat{\mu}' = \hat{\mu}[l+i \mapsto (\hat{v}, k'_l, k'_r)]}{\mathbf{store}(\hat{\mu}, l, \hat{\sigma}, \hat{v}) \rightsquigarrow_s \mathbb{S}\langle \hat{\mu}', \pi' \rangle}
\end{array}$$

■ **Figure 5** Core Semantics: Memory Actions.

In order to define the symbolic semantics of our core language, we make use of computation *outcomes* [21, 36], which capture the flow of execution and are generated by the following grammar: $\hat{\sigma} \in \hat{\mathcal{O}} ::= \mathbb{C}\langle \hat{s} \rangle \mid \mathbb{C}\langle \cdot \rangle \mid \mathbb{R}\langle \hat{v} \rangle \mid \mathbb{E}\langle \pi \rangle$. We make use of four types of outcomes: **(1)** the non-empty continuation outcome $\mathbb{C}\langle \hat{s} \rangle$, signifying that the execution of the current statement generated a new statement to be executed next; **(2)** the empty continuation outcome $\mathbb{C}\langle \cdot \rangle$, signifying that the execution may proceed to the next instruction; **(3)** the return continuation outcome $\mathbb{R}\langle \hat{v} \rangle$, signifying that the current execution terminated with return value \hat{v} ; and **(4)** the error outcome $\mathbb{E}\langle \pi \rangle$, signifying that the current execution generates an error.

We define the symbolic semantics of our core language using a semantic judgement of the form: $\hat{\sigma}, s \rightsquigarrow \hat{\sigma}', \hat{\sigma}$, meaning that the symbolic evaluation of statement s in the state $\hat{\sigma}$ generates the state $\hat{\sigma}'$ and outcome $\hat{\sigma}$. We splice the components of the state into the semantic transition, simply writing $\langle \hat{\mu}, \hat{\rho}, \pi, s \rangle \rightsquigarrow \langle \hat{\mu}', \hat{\rho}', \pi', \hat{\sigma} \rangle$ when $\hat{\sigma} = (\hat{\mu}, \hat{\rho}, \pi)$ and $\hat{\sigma}' = (\hat{\mu}', \hat{\rho}', \pi')$. The semantic rules are given in Figures 4 and 6, where the former focus on the core language and the latter on symbolic reflection API. Due to space constraints, we omit all error-generating transitions with the exception of those describing errors generated by API primitives. We further omit the elements of the configuration that are neither updated nor inspected by the current rule, writing, for instance, $\hat{\rho}, s \rightsquigarrow \hat{\rho}', \hat{\sigma}$ to mean $\langle \hat{\mu}, \hat{\rho}, \pi, s \rangle \rightsquigarrow \langle \hat{\mu}, \hat{\rho}', \pi, \hat{\sigma} \rangle$. Note that the semantics is non-deterministic, meaning the symbolic execution of a statement on a given state may generate multiple states and continuations.

The symbolic semantics of the imperative fragment is straightforward. It makes use of the auxiliary function **alloc** and relations **load** and **store** for interacting with the linear memory; their meanings are as follows:

- **alloc**($\hat{\mu}, n$) allocates a new memory block of size n ;
- **load**($\hat{\mu}, l, \hat{\sigma}$) $\rightsquigarrow_s \mathbb{S}\langle \hat{v}, \pi' \rangle$ successfully loads the value \hat{v} stored at the offset $\hat{\sigma}$ of location l in memory $\hat{\mu}$; as the loading operation may cause the execution to branch, it additionally generates a new formula π' with the conditions that must hold for the symbolic value \hat{v} to be returned;
- **store**($\hat{\mu}, l, \hat{\sigma}, \hat{v}$) $\rightsquigarrow \mathbb{S}\langle \hat{\mu}', \pi' \rangle$ successfully stores the symbolic value \hat{v} at the offset $\hat{\sigma}$ of location l in memory $\hat{\mu}$ under the path condition π , returning a new memory $\hat{\mu}'$; as for the loading operation, the storing operation may cause the execution to branch, hence the returned constraint π' .

In the above, we use the symbol \mathbb{S} to distinguish the successful transitions of **load** and **store** from their error-leading transitions, which are labelled with \mathbb{E} .

The definitions of **alloc**, **load**, and **store** are given in Figure 5. In contrast to **load** and **store** which may cause the current execution to branch, **alloc** is assumed to be deterministic. Hence, it is modelled as a function. Load and store operations *fail* if they attempt to read/update the contents of a memory cell beyond the bounds of the inspected

$$\begin{array}{c}
\text{ASSERT - FALSE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \pi' \quad \not\vdash \pi \Rightarrow \pi'}{\hat{\rho}, \pi, \mathbf{assert}(e) \rightsquigarrow \hat{\rho}, \pi, \mathbb{E}(\pi)} \\
\\
\text{ASSERT - TRUE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \pi' \quad \vdash \pi \Rightarrow \pi'}{\hat{\rho}, \pi, \mathbf{assert}(e) \rightsquigarrow \hat{\rho}, \pi, \mathbb{C}(\cdot)} \\
\\
\text{ASSUME} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \pi'}{\hat{\rho}, \pi, \mathbf{assume}(e) \rightsquigarrow \hat{\rho}, \pi \wedge \pi', \mathbb{C}(\cdot)} \\
\\
\text{ISSYMBOLIC - TRUE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} \notin \mathit{Consts} \quad \hat{\rho}' = \hat{\rho}[x \mapsto \mathit{true}]}{\hat{\rho}, x := \mathbf{is_symbolic}(e) \rightsquigarrow \hat{\rho}', \mathbb{C}(\cdot)} \\
\\
\text{ISSYMBOLIC - FALSE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} \in \mathit{Consts} \quad \hat{\rho}' = \hat{\rho}[x \mapsto \mathit{false}]}{\hat{\rho}, x := \mathbf{is_symbolic}(e) \rightsquigarrow \hat{\rho}', \mathbb{C}(\cdot)} \\
\\
\text{SYMB} \\
\frac{\hat{x} \text{ fresh}}{\hat{\rho}, x := \mathbf{symb}() \rightsquigarrow \hat{\rho}[x \mapsto \hat{x}], \mathbb{C}(\cdot)} \\
\\
\text{ISSAT - TRUE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \pi' \quad \pi \wedge \pi' \text{ SAT} \quad \hat{\rho}' = \hat{\rho}[x \mapsto \mathit{true}]}{\hat{\rho}, \pi, x := \mathbf{is_sat}(e) \rightsquigarrow \hat{\rho}', \pi, \mathbb{C}(\cdot)} \\
\\
\text{ISSAT - FALSE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \pi' \quad \pi \wedge \pi' \text{ UNSAT} \quad \hat{\rho}' = \hat{\rho}[x \mapsto \mathit{false}]}{\hat{\rho}, \pi, x := \mathbf{is_sat}(e) \rightsquigarrow \hat{\rho}', \pi, \mathbb{C}(\cdot)} \\
\\
\text{MAXIMISE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \hat{e} \quad \pi \wedge \hat{e} \leq v \text{ SAT} \quad \pi \wedge \hat{e} > v \text{ UNSAT} \quad \hat{\rho}' = \hat{\rho}[x \mapsto v]}{\hat{\rho}, \pi, x := \mathbf{maximize}(e) \rightsquigarrow \hat{\rho}', \pi, \mathbb{C}(\cdot)} \\
\\
\text{MINIMISE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \hat{e} \quad \pi \wedge \hat{e} \geq v \text{ SAT} \quad \pi \wedge \hat{e} < v \text{ UNSAT} \quad \hat{\rho}' = \hat{\rho}[x \mapsto v]}{\hat{\rho}, \pi, x := \mathbf{minimize}(e) \rightsquigarrow \hat{\rho}', \pi, \mathbb{C}(\cdot)} \\
\\
\text{CUR-PC} \\
\frac{\hat{\rho}' = \hat{\rho}[x \mapsto \pi]}{\hat{\rho}, \pi, x := \mathbf{cur_pc}() \rightsquigarrow \hat{\rho}', \pi, \mathbb{C}(\cdot)} \\
\\
\text{BLOCKSIZE} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = l \quad \mathbf{b_size}(\hat{\mu}, l) = k \quad \hat{\rho}' = \hat{\rho}[x \mapsto k]}{\hat{\rho}, \pi, x := \mathbf{block_size}(e) \rightsquigarrow \hat{\rho}', \pi, \mathbb{C}(\cdot)} \\
\\
\text{NOT} \\
\frac{\llbracket e \rrbracket_{\hat{\rho}} = \hat{e}' \quad \hat{\rho}' = \hat{\rho}[x \mapsto \mathit{uop}(\hat{e}')] }{\hat{\rho}, x := \mathbf{constr}_{\mathit{uop}}(e) \rightsquigarrow \hat{\rho}', \mathbb{C}(\cdot)} \\
\\
\text{BI-CONTR} \\
\frac{\llbracket e_i \rrbracket_{\hat{\rho}} = \hat{e}_i |_{i=1,2} \quad \hat{\rho}' = \hat{\rho}[x \mapsto \mathit{bop}(\hat{e}_1, \hat{e}_2)] }{\hat{\rho}, x := \mathbf{constr}_{\mathit{bop}}(e_1, e_2) \rightsquigarrow \hat{\rho}', \mathbb{C}(\cdot)}
\end{array}$$

■ **Figure 6** Core Semantics: Symbolic Reflection API.

location (the failing rules are omitted due to space constraints). In the success case, both load and store operations branch on the value of all legal offsets. This means that these operations may generate unsatisfiable path conditions (for instance, when the given offset is concrete), in such cases the symbolic execution path is unfeasible and will be filtered out by symbolic execution.

Finally, Figure 6 gives the rules that describe the semantics of our proposed API. The rules are straightforward. Note that constraints are simply symbolic values of boolean type; hence, various rules either assign constraints to variables (e.g. CUR-PC) or obtain a constraint as the result of symbolically evaluating an expression (e.g. ASSERT, ASSUME, ISSAT).

Symbolic Execution – Collecting Semantics

So far, we have defined the semantics of a single symbolic execution trace. In the following, we extend this definition to account for multiple traces. We use $\hat{\phi}$ and $\hat{\omega}$ to range over input and output symbolic configurations, respectively.² We further use $\hat{\Omega}$ to range over sets of

² Input configurations differ from output configurations in that former are composed of a symbolic state and a statement whereas the latter are composed of a symbolic state and an outcome.

output configurations. We capture the semantics of multiple-trace symbolic execution using a big-step relation of the form $\hat{\phi} \Downarrow \hat{\Omega}$, meaning that if we “run” the symbolic semantics on the input configuration $\hat{\phi}$, we obtain the set of output configurations $\hat{\Omega}$. As symbolic execution often diverges, we additionally introduce a bounded version of the collecting semantics, writing $\hat{\phi} \Downarrow_k \hat{\Omega}$ to mean that if we “run” the symbolic semantics on the input configuration $\hat{\phi}$ and ignore symbolic traces with more than k steps, we end up with the set of output configurations $\hat{\Omega}$. In order to formalise these relations, we make use of collecting one-step transitions, writing $\hat{\phi} \searrow \hat{\Omega}$ to mean $\hat{\Omega}$ contains all the configurations resulting from the application of a single symbolic step on the input configuration $\hat{\phi}$ and no other; put formally:

$$\hat{\Omega} = \frac{\{\langle \hat{\mu}', \hat{\rho}', \pi', \hat{\delta} \rangle \mid \langle \hat{\mu}, \hat{\rho}, \pi, \hat{s} \rangle \rightsquigarrow \langle \hat{\mu}', \hat{\rho}', \pi', \hat{\delta} \rangle\}}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s} \rangle \searrow \hat{\Omega}}$$

Definition 1 formalises the collecting semantics for bounded symbolic execution (the unbounded version can be easily obtained from the bounded one by dropping the constraints on k). The definition makes use of the following auxiliary predicates and functions:

- The predicates **Final**($\hat{\omega}$) and **NonFinal**($\hat{\omega}$) respectively hold if the output configuration $\hat{\omega}$ is, respectively, final and non-final, with a configuration being final if it contains either a return or an error outcome.
- The function **Next** can only be applied to non-final output configurations, turning the given output configuration into an input configuration by unwrapping the statement contained in its non-empty continuation output.

We say that **Final/NonFinal** holds for a set of configurations if it holds for all of them.

► **Definition 1** (Symbolic Execution – Collecting Semantics).

$$\frac{\text{BOUNDED - BASE} \quad \langle \hat{\mu}, \hat{\rho}, \pi, \hat{s} \rangle \searrow \hat{\Omega}_1 \cup \hat{\Omega}_2 \quad \pi \text{ SAT} \quad \text{Final}(\hat{\Omega}_1) \quad \text{NonFinal}(\hat{\Omega}_2)}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s} \rangle \Downarrow_1 \hat{\Omega}_1} \quad \frac{\text{BOUNDED - FALSE} \quad \pi \text{ UNSAT}}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s} \rangle \Downarrow_k \emptyset}$$

$$\frac{\text{BOUNDED - REC} \quad \langle \hat{\mu}, \hat{\rho}, \pi, \hat{s} \rangle \searrow \hat{\Omega} \cup \{\hat{\omega}_i \mid_{i=1}^n\} \quad \text{Final}(\hat{\Omega}) \quad k > 1 \quad \pi \text{ SAT} \quad \text{Next}(\hat{\omega}_i) \Downarrow_{k-1} \hat{\Omega}_i \mid_{i=1}^n}{\langle \hat{\mu}, \hat{\rho}, \pi, \hat{s} \rangle \Downarrow_k \cup_{i=1}^n \hat{\Omega}_i \cup \hat{\Omega}}$$

3.2 Summary Correctness Properties

In contrast to most works on verification in which a concrete program is proven correct with respect to a specification [5], here we take the concrete function associated with the given summary to be the ground truth. Given a summary \hat{s} and its associated concrete implementation s , we say that:

- \hat{s} is an *over-approximation* of s if all the concrete executions of s are contained in the set of the executions modelled by \hat{s} ;
- \hat{s} is an *under-approximation* of s if all the executions modelled by \hat{s} are contained in the set of concrete executions of s .

When a summary satisfies both properties, we say that it is *exact*. In the following, we make use of the concrete and symbolic semantics of our core language to establish the rigorous definitions underpinning these concepts.

Symbolic State Interpretation

We write $\sigma \in \llbracket \hat{\sigma} \rrbracket$ to mean that the concrete state σ is in the interpretation of the symbolic state $\hat{\sigma}$. The interpretation of a symbolic state $\hat{\sigma}$ is the set of concrete states that can be obtained from $\hat{\sigma}$ by mapping the symbolic values of $\hat{\sigma}$ to concrete values in a way that is

consistent with its path condition. For instance, if $\hat{\sigma} = \langle \hat{\mu}, \hat{\rho}, \hat{x} \neq 0 \rangle$, then the symbolic variable \hat{x} cannot be replaced by 0 in $\hat{\mu}$ and $\hat{\rho}$. Accordingly, the interpretation function $\llbracket \cdot \rrbracket :: \text{SymSt} \rightarrow \wp(\text{ConcSt})$ takes as input a symbolic state and returns a set of concrete states. We define the interpretation function for symbolic states with the help of two auxiliary interpretation functions, one for symbolic memories and one for symbolic stores. These interpretation functions require a *valuation function*, $\varepsilon : \hat{\mathcal{X}} \rightarrow \mathcal{V}$, that maps symbolic variables to concrete variables. We write $\llbracket \hat{\mu} \rrbracket_\varepsilon = \mu$ to mean that the interpretation of $\hat{\mu}$ under ε yields the concrete memory μ (analogously for stores). We interpret symbolic memories and stores point-wise, applying the valuation function to each memory/store cell. Put formally:

$$\begin{array}{ccc} \text{HEAP-INTERP.} & \text{STORE-INTERP.} & \text{STATE-INTERP.} \\ \frac{\hat{\mu}(l) = (\hat{v}, k_l, k_r)}{\llbracket \hat{\mu} \rrbracket_\varepsilon(l) \triangleq (\llbracket \hat{v} \rrbracket_\varepsilon, k_l, k_r)} & \frac{\hat{\rho}(x) = \hat{v}}{\llbracket \hat{\rho} \rrbracket_\varepsilon(x) \triangleq \llbracket \hat{v} \rrbracket_\varepsilon} & \frac{\hat{\sigma} = \langle \hat{\mu}, \hat{\rho}, \pi \rangle}{\llbracket \hat{\sigma} \rrbracket \triangleq \{ (\llbracket \hat{\mu} \rrbracket_\varepsilon, \llbracket \hat{\rho} \rrbracket_\varepsilon) \mid \llbracket \pi \rrbracket_\varepsilon = \text{true} \}} \end{array}$$

In the definitions that follow, we characterise the correctness of a given summary with respect to its corresponding reference implementation. In this context, the contents of the store are not relevant as they are discarded after the execution of the function. To account for this, we make use of *truncated state interpretations*, which ignore the store component. We use $\llbracket \hat{\sigma} \rrbracket$ to refer to the truncated interpretation of the symbolic state $\hat{\sigma}$, which is defined as follows: $\llbracket \langle \hat{\mu}, \hat{\rho}, \pi \rangle \rrbracket \triangleq \{ (\llbracket \hat{\mu} \rrbracket_\varepsilon \mid \llbracket \pi \rrbracket_\varepsilon = \text{true}) \}$. For convenience, we lift truncated symbolic state interpretation to pairs of symbolic states and symbolic outcomes as follows: $\llbracket (\langle \hat{\mu}, \hat{\rho}, \pi \rangle, \hat{o}) \rrbracket \triangleq \{ (\llbracket \hat{\mu} \rrbracket_\varepsilon, \llbracket \hat{o} \rrbracket_\varepsilon) \mid \llbracket \pi \rrbracket_\varepsilon = \text{true} \}$.

Correctness Properties

We are now at a position to formally define the correctness properties of symbolic summaries. Definitions 2 and 3 respectively define over- and under-approximating summaries. We omit the definition of *exactness*, which is simply the conjunction of the first two. Both definitions rely on the concrete semantics of our core language, using $\langle \mu, \rho, s \rangle \rightarrow_k \langle \mu', \rho', o \rangle$ to state that the concrete execution of s in the concrete memory μ and store ρ , finishes in k steps and generates the concrete memory μ' , store ρ' , and outcome o .

► **Definition 2 (Bounded Over-Approximating Summary).** *A symbolic summary \hat{s} is said to be a k -bound over-approximation of a concrete implementation s with respect to a symbolic memory $\hat{\mu}$ and symbolic store $\hat{\rho}$, if and only if it holds that:*

$$\begin{aligned} \langle \hat{\mu}, \hat{\rho}, \text{true}, \hat{s} \rangle &\rightsquigarrow^* \langle \hat{\mu}', \hat{\rho}', \pi, \hat{o} \rangle \wedge \text{Final}(\hat{o}) \\ \implies \forall \mu, \mu', \rho, \rho', o, k'. (\mu, \rho) \in \llbracket \langle \hat{\mu}, \hat{\rho}, \pi \rangle \rrbracket &\wedge \langle \mu, \rho, s \rangle \rightarrow_{k'} \langle \mu', \rho', o \rangle \wedge k' \leq k \wedge \text{Final}(o) \\ \implies (\mu', o) \in \llbracket \langle \hat{\mu}', \hat{\rho}', \pi, \hat{o} \rangle \rrbracket & \end{aligned}$$

► **Definition 3 (Under-Approximating Summary).** *A symbolic summary \hat{s} is said to be an under-approximation of a concrete implementation s with respect to a symbolic memory $\hat{\mu}$ and symbolic store $\hat{\rho}$, if and only if it holds that:*

$$\begin{aligned} \langle \hat{\mu}, \hat{\rho}, \text{true}, \hat{s} \rangle &\rightsquigarrow^* \langle \hat{\mu}', \hat{\rho}', \pi, \hat{o} \rangle \wedge \text{Final}(\hat{o}) \\ \implies \forall \mu', o. (\mu', o) \in \llbracket \langle \hat{\mu}', \hat{\rho}', \pi, \hat{o} \rangle \rrbracket & \\ \implies \exists \mu, \rho, \rho'. (\mu, \rho) \in \llbracket \langle \hat{\mu}, \hat{\rho}, \pi \rangle \rrbracket &\wedge \langle \mu, \rho, s \rangle \rightarrow^* \langle \mu', \rho', o \rangle \end{aligned}$$

The proposed definitions are unusual in that they relate the symbolic execution of the summary \hat{s} with the concrete executions of its reference implementation s . Typical definitions of this type [21, 36, 48, 47] relate symbolic execution of a given program with its concrete execution. In our setting, we have two different programs being related: a symbolic summary and its reference implementation.

■ **Table 1** C-Implemented summaries.

Class	N	Lines of code	API calls	Memory	Effects	Symbolic	Return
				×	✓	×	✓
<i>Strings</i>	34	1639	194	24	10	18	16
<i>Number Parsing</i>	6	563	56	6	0	0	6
<i>I/O</i>	6	79	16	4	2	5	1
<i>Memory</i>	14	508	71	8	6	9	5
<i>Heap</i>	7	174	29	5	2	7	0
Total	67	2814	344	47	20	39	28

Finally, we note that the proposed definitions require that over/under-approximating summaries allocate memory in the exact same order of their corresponding reference implementations. This requirement could be relaxed by modifying our definition of symbolic heap interpretation for it to relate each symbolic heap with all its valid concrete reshufflings instead of simply those that follow its allocation order. We opted, however, for the more restrictive definitions in order to simplify the presentation.

3.3 Modelling LIBC Functions

We illustrate how the symbolic reflection API can be used to implement symbolic summaries for two families of LIBC functions: string manipulation functions and number-parsing functions. In total, we have implemented 67 summaries targeting 24 LIBC functions. Table 1 gives an overview of the 67 implemented summaries, showcasing for each category of summaries: (1) the total number of lines of code; (2) the total number of calls to the symbolic reflection API; (3) the number of summaries that update the heap memory; and (4) the number of summaries that return symbolic values. In the following, we give an example of a summary from each category.

Example – String Summaries

Figure 7 shows the implementation of an under-approximating symbolic summary for the `strlen` function (recall that the summary given in Figure 2 is exact). This summary iterates over an input string until it finds a concrete null character. During this process, if it finds a symbolic character, it tries to prove that the corresponding byte can only be a null character. If it succeeds, then the summary returns the current length, otherwise it assumes that the current character is not the null character and continues iterating. More concretely, if a character `s[i]` is symbolic, the summary builds the constraint `cnstr ≡ (s[i] ≠ \0)` and uses the primitive `is_sat` to check if `s[i]` can only be the null character (i.e., `!is_sat(cnstr)`), in which case the summary simply returns the value of `i`. Otherwise, `cnstr` is added to the current path condition using the primitive `assume`, making the summary under-approximating. For example, given the symbolic string `[c0, 'a', c2, \0]`, where `c0` and `c2` denote unconstrained symbolic characters, the summary outputs the value 3 and adds the constraints `c0 ≠ \0` and `c2 ≠ \0` to the path condition.

```

1 int strlen2(char* s){
2   char charZero = '\0'; int i = 0;
3   while(1){
4     if(is_symbolic(&s[i])){ //s[i] is symbolic
5       cnstr_t cnstr = _solver_NEQ(&s[i], &charZero, CHAR_SIZE); // s[i] ≠ '\0'
6       if(!is_sat(cnstr)) break;
7       else assume(cnstr); //Add cnstr to symbolic state
8     }
9     else if(s[i] == charZero) break;
10    i++;
11  }
12  return i;
13 }

```

■ **Figure 7** Under-approximating Summary of `strlen`.

```

1 int atoi2(char *str){
2   ...
33  else {
34    symbolic retval = new_sym_var(INT_SIZE);
35    int size = strlen(str); //Max possible length
36
37    //Determine bounds
38    int lower_bound = pow(10,size-1) * -1;
39    int upper_bound = pow(10,size);
40
41    //Build interval with constraints
42    cnstr_t val_GT_lower = _solver_SGT(&retval, &lower_bound, INT_SIZE);
43    cnstr_t val_LT_upper = _solver_SLT(&retval, &upper_bound, INT_SIZE);
44    cnstr_t bounds_cnstr = _solver_And(val_GT_lower,val_LT_upper);
45
46    //Add constraints to symbolic state
47    assume(bounds_cnstr);
48    return retval;
49  }
50 }
51 return res * sign;
52 }

```

■ **Figure 8** Fragment of Over-approximating Summary of `atoi`.

Example – Number summaries

Figure 8 shows a fragment of an over-approximating summary for `atoi`. The `atoi` function is used to parse strings denoting integer values. The key for guaranteeing that the summary is over-approximating is to return a fresh symbolic value and constrain this value to be: (i) greater than or equal to the smallest possible value that may be denoted by the given string; and (ii) smaller than or equal to the largest possible value that may be denoted by the given string. To this end, the summary determines the maximum possible length of the given string and uses it to compute the interval of possible return values. For example, considering a symbolic string `[c0, c1, c2, \0]`, the summary constrains the returned symbolic value, `retval` to lie within the interval `retval ∈ [−99, 999]`.

3.4 Reflection API Implementation

The proposed API can be implemented on top of any symbolic execution tool whose representation of symbolic states includes a symbolic memory and a path condition. We have found this to be the case for all the symbolic execution tools that we have analysed so

far [50, 19, 8, 15, 38, 43, 17, 16, 31]. In the formalism we have chosen to include a symbolic store component, to simplify the presentation, but this component is not essential for implementing the API.

In order to illustrate the effort involved in implementing the API, we have extended *AVD* [45] and *angr* [50], two current symbolic execution tools, with support for it. *AVD* is a novel symbolic execution tool developed by the authors, whereas *angr* is a widely used binary analysis toolkit. In both cases, the API implementation was straightforward, with the API code totalling 330 LoC for *angr* and 529 LoC for *AVD*. Table 2 shows the number of LoC of both implementations per type of API primitive. API primitives were implemented differently in *AVD* and *angr*. In *AVD*, we have implemented each reflection primitive as if it were a native symbolic summary, interacting directly with our own internal representation of symbolic states. In *angr*, we have used an internal API provided by the tool for developers to implement their own *angr* summaries directly in Python, associating each API primitive to an *angr simprocedure*.

■ **Table 2** Lines of code per type of API primitive implemented in *AVD* and *angr*.

	Reflection Primitives				Total
	Core	Memory	Symbolic Values	Constraints	
AVD	206	54	56	213	529
angr	100	23	53	154	330

We believe that extending other symbolic execution tools with support for the proposed API should be as straightforward as extending *angr* and *AVD* since most tools already possess internally, albeit with variations, the reflection mechanisms captured by our API. The main difficulty in the implementation of the API is that it requires a thorough understanding of the inner-workings of the targeted tool and therefore should be done by the tools' own engineering teams rather than by external users. This effort should, however, pay off as implementing our API requires considerably less code than implementing a comprehensive library of symbolic summaries for LIBC from scratch, while also being conceptually simpler.

4 SumBoundVerify: Bounded Verification of Symbolic Summaries

In this section, we introduce our proposed methodology for the bounded verification of symbolic summaries. We first introduce our main summary verification algorithm (§4.1) and then explain how we leverage this algorithm to build `SUMBOUNDVERIFY` by automatically generating symbolic tests for the summaries to be verified (§4.2).

4.1 Bounded Summary Verification Algorithm

Symbolic State Lifting

In order to verify the correctness properties of a summary, we introduce a lifting operator $[\cdot] :: \mathcal{P}(\text{SymSt} \times \hat{\mathcal{O}}) \rightarrow \Pi$ that transforms a set of output configurations (i.e. symbolic states paired up with symbolic outcomes) into a boolean formula. We write $[\hat{\Omega}] = \pi$ to denote that the lifting of the output configurations in $\hat{\Omega}$ generates the formula π . The lifting operator is formally defined as follows:

$$[\hat{\Omega}] \triangleq \bigvee \left\{ [\hat{\mu}]_m \wedge \pi \wedge (\text{ret} = \hat{v}) \mid (\langle \hat{\mu}, \hat{\rho}, \pi \rangle, \mathbb{R}\langle \hat{v} \rangle) \in \hat{\Omega} \right\}$$

24:16 Toward Tool-Independent Summaries for Symbolic Execution

Essentially, a set of output configurations is transformed into a disjunction of boolean formulas, each describing its corresponding configuration. The formula created for each configuration has three components: (i) a memory component $[\hat{\mu}]_m$ describing the content of the symbolic memory (defined below); (ii) a path condition component π ; and (iii) a return component $\mathbf{ret} = \hat{v}$, describing the return value of the function in the execution path that led to the given configuration. We use two dedicated variable \mathbf{ret} and \mathbf{count} to refer to the return value of a function and the number of cells in the current heap, respectively. Importantly, the lifting of a set of output configurations is only defined if all configurations are associated with a return outcome. The lifting operator for symbolic memories $[\cdot]_m :: \mathit{SymMem} \rightarrow \Pi$, which takes a symbolic memory $\hat{\mu}$ and returns a boolean formula $[\hat{\mu}]_m$ describing its contents, is defined as follows:

$$\frac{\text{MEMORY LIFTING} \quad \pi_{\mathit{blocks}} = \bigwedge_{l \in \mathit{blocks}(\hat{\mu})} [\hat{\mu}]_b^l \quad \pi_{\mathit{count}} = \mathbf{count} = |\mathbf{dom}(\hat{\mu})|}{[\hat{\mu}]_m \triangleq \pi_{\mathit{blocks}} \wedge \pi_{\mathit{count}}} \quad \frac{\text{BLOCK LIFTING} \quad \hat{\mu}(l) = (-, 0, k_r)}{[\hat{\mu}]_b^l \triangleq \bigwedge_{0 \leq i < k_r} \{\mathbf{cell}(l+i, \hat{\mu}(l+i))\}}$$

A symbolic memory $\hat{\mu}$ is encoded into the conjunction of its blocks, which are, in turn, encoded using an auxiliary encoding function $[\cdot]_b : \mathit{SymMem} \times \mathbb{N} \rightarrow \Pi$ for lifting memory blocks to formulas. The memory encoding function makes use of an auxiliary function $\mathbf{blocks} : \mathit{SymMem} \rightarrow \wp(\mathbb{N})$ to obtain the locations in the given memory corresponding to the beginning of blocks. The block-lifting function transforms each memory block into a formula describing the contents of each cell in the given block. To this end, we make use of an uninterpreted predicate \mathbf{cell} to denote that the cell at a given address has the given content.

Bounded Verification Algorithm

Algorithm 1 describes our procedure for verifying if a summary \hat{s} is under/over-approximating with respect to a concrete implementation s , symbolic memory $\hat{\mu}$, and symbolic store $\hat{\rho}$. In a nutshell, this algorithm compares the symbolic state(s) resulting from the execution of the summary, $\hat{\Omega}_{\mathit{sum}}$, against those generated by its reference implementation, $\hat{\Omega}_{\mathit{ref}}$. We do not bound the symbolic execution of the summary given that summaries should be designed to be convergent. In contrast, we bound the execution of their reference implementations as they often diverge. In a nutshell, we conclude that:

- a summary is *over-approximating* if $[\hat{\Omega}_{\mathit{ref}}] \implies [\hat{\Omega}_{\mathit{sum}}]$, i.e. the set of symbolic states generated by the reference implementation are contained in those generated by the summary;
- a summary is *under-approximating* if $[\hat{\Omega}_{\mathit{sum}}] \implies [\hat{\Omega}_{\mathit{ref}}]$, i.e. the set of symbolic states generated by the summary are contained in those generated by the corresponding reference implementation.

These implications are, however, too strong as they do not account for the creation of new symbolic values within the execution of the summary. To account for these, we have to existentially quantify the symbolic variables created during the execution of the summary. Algorithm 1 makes use of an auxiliary function $\mathbf{existentials}$ for computing the variables to be existentially quantified according to the following equation:

$$\mathbf{existentials}(\pi, \hat{\mu}, \hat{\rho}) \triangleq \mathbf{lvars}(\pi) \setminus (\mathbf{lvars}(\hat{\mu}) \cup \mathbf{lvars}(\hat{\rho}))$$

Essentially, all the symbolic variables created during the execution of the summary must be existentially quantified; these correspond to the symbolic variables that exist in the final symbolic states obtained from the summary execution but do not in the initial state.

Algorithm 1 Bounded Summary Verification.

```

1 function VERIFYSUMMARY( $prop, \hat{s}, s, \hat{\mu}, \hat{\rho}, k$ )
2    $\langle \hat{\mu}, \hat{\rho}, true, \hat{s} \rangle \Downarrow \hat{\Omega}_{sum}$ 
3    $\langle \hat{\mu}, \hat{\rho}, true, s \rangle \Downarrow_k \hat{\Omega}_{ref}$ 
4    $\pi_{sum} \leftarrow \lceil \hat{\Omega}_{sum} \rceil$ 
5    $\pi_{ref} \leftarrow \lceil \hat{\Omega}_{ref} \rceil$ 
6    $xs_{sum} \leftarrow \text{existentials}(\pi_{sum}, \hat{\mu}, \hat{\rho})$ 
7   if  $prop = over$  then
8      $out \leftarrow \text{isValid}(\pi_{ref} \implies \exists xs_{sum}. \pi_{sum})$ 
9   else
10     $out \leftarrow \text{isValid}(\exists xs_{sum}. \pi_{sum} \implies \pi_{ref})$ 
11  return  $out$ 

```

Correctness Result

Theorem 4 is our main correctness result. Essentially, it states that if we apply VERIFYSUMMARY in over-approximation mode and it returns *true*, then the given summary is a bounded over-approximation of the given concrete implementation and analogously for under-approximation mode.

► **Theorem 4** (Summary Correctness). *Let \hat{s} be a symbolic summary, s its reference implementation, $\hat{\mu}$ a symbolic memory, $\hat{\rho}$ a symbolic store, and k and a positive integer; then, it holds that:*

- *If $\text{VERIFYSUMMARY}(over, \hat{s}, s, \hat{\mu}, \hat{\rho}, k) = true$, then \hat{s} is a k -bound over-approximation of s with respect to $\hat{\mu}$ and $\hat{\rho}$;*
- *If $\text{VERIFYSUMMARY}(under, \hat{s}, s, \hat{\mu}, \hat{\rho}, k) = true$, then \hat{s} is an under-approximation of s with respect to $\hat{\mu}$ and $\hat{\rho}$.*

4.2 Automatic Symbolic Test Generation

In §4.1, we introduced a method for checking whether or not a given summary \hat{s} is correct with respect to a reference implementation s , a symbolic memory $\hat{\mu}$, and a symbolic store $\hat{\rho}$. Naturally, one would like to prove that a summary is correct with respect to all symbolic memories and stores consistent with the signature of the summarised function instead of only a particular symbolic memory and store. SUMBOUNDVERIFY solves this problem partially by bounding the size of memories and stores to be explored and using the type information in the signature of the summarised function to generate the initial symbolic states for which to check the summary up to the pre-established bound. In this section, we explain the procedure by example, leaving its formalisation for future work.

Instead of directly creating the symbolic states on which to run the summaries to be evaluated, we generate the initialisation code that creates such states from the signature of the function to be summarised. In general, the generated initialisation code depends on the type of the arguments given to the summary:

- For *non-character arrays*, we generate one fully symbolic array for each array size under the specified bound;
- For *character arrays*, we generate a single fully symbolic array terminated with a concrete null character with size given by the specified bound (note that this single character array models all strings of size lower than the specified bound since the intermediate symbolic characters of the array may denote the null character);

24:18 Toward Tool-Independent Summaries for Symbolic Execution

- For *non-recursive structures*, we generate a single fully symbolic test with the elements of the structure being mapped to fresh symbolic values;
- For *recursive structures*, we generate one fully symbolic test for each unfolding of the recursive structure up to the specified bound.

The test generation algorithm has two important limitations. First, it does not cover cases in which there are mutual dependencies between the parameters of the function to be summarised (e.g. a function with two array parameters with shared elements). Second, when it comes to recursive structures, the algorithm does not consider structures with loops, such as doubly-linked lists. If the arguments of the summarised function may exhibit one of these features, then the corresponding tests should be created manually.

Examples - Non-Character vs. Character Arrays

We now illustrate the test generation algorithm with two simple examples, covering non-character and character arrays. Suppose we want to validate a summary for a function with signature `int f(int* arr)`; in this case, `SUMBOUNDVERIFY` would generate a set of tests, each with an initial section in charge of creating the symbolic integer array associated with the parameter `arr`. An example of one such initialisation code is given below.

```
1 int arr[4];
2 for (int i = 0; i < 4; i++) { arr[i] = new_sym_var_array("i", i, INT_SIZE); }
```

Essentially, the initialisation code allocates an integer array of size 4 in the stack and fills the four elements of the array with fresh symbolic integers. Suppose now, we want to validate a summary for a function with signature `int g(char* s)`; in this case, `SUMBOUNDVERIFY` would generate a single test with the initialisation code given below.

```
1 char s[BOUND];
2 for (int i = 0; i < BOUND-1; i++) { s[i] = new_sym_var_array("c", i, CHAR_SIZE); }
3 s[BOUND-1] = '\0';
```

The initialisation code for character arrays has two main differences with respect to the code generated for non-character arrays: **(1)** the size of the character array allocated in the stack is always set to the specified bound and **(2)** the last element of the allocated character array is always set to the concrete null character.

5 Evaluation

This section answers the following evaluation questions:

- EQ1** - Is the proposed symbolic reflection API sufficiently expressive to allow for the implementation of under/over-approximating summaries?
- EQ2** - Can tool independent symbolic summaries be used to contain path explosion in symbolic execution?
- EQ3** - Can `SUMBOUNDVERIFY` be used to analyse real-world symbolic summaries developed in the context of state-of-the-art symbolic execution tools?

5.1 EQ1: API Expressivity

In order to illustrate the expressivity of our symbolic reflection API, we implement a library of symbolic summaries consisting of 67 summaries covering 26 LIBC functions from three different header files: `string.h`, `stdlib.h` and `stdio.h`. For most of these functions, we have implemented

at least two summaries, each illustrating a different correctness property. We then used `SUMBOUNDVERIFY` to check the correctness properties of the implemented summaries by comparing them against their corresponding reference implementations. The fact that we were able to implement both under/over-approximating summaries for a large number of LIBC functions gives us a strong confidence that our library is expressive enough to model a wide range of behaviours.

Table 3 shows, for each function category, the number of implemented summaries and their corresponding correctness properties. In the table, N represents the total number of implemented summaries; N/A represents the number of implemented summaries that do not satisfy any correctness property; and the remaining columns represent the number of implemented summaries that satisfy the corresponding property (Under, Over, or Exact).

■ **Table 3** Correctness properties for the C-implemented summaries.

Category	N	N/A	<i>Under-approx.</i>	<i>Over-approx.</i>	<i>Exact</i>
<i>Strings</i>	34	14	7	6	7
<i>Memory</i>	14	6	4	2	2
<i>Number Parsing</i>	6	2	–	4	–
<i>System Calls</i>	13	13	–	–	–
Total	67	35	11	12	9

Note that the summaries modelling functions that use system calls (e.g., `malloc` and `fgets`) cannot be verified against their respective reference implementations. The reason is that the symbolic execution of the reference implementations would necessarily step outside the perimeter of the language and, therefore, of the symbolic execution engine. For instance, the function `fgets` uses the `read` system call to obtain the string with the contents of the given file. In order to symbolically execute `fgets`, we always need to have a summary of `read` to start with, and this bootstrapping summary cannot be symbolically checked. Additionally, some summaries are neither over- nor under-approximating for performance reasons. These summaries assume that the function inputs satisfy certain preconditions, which we explicitly specify as comments in the summary code. The formal characterisation and verification of the properties satisfied by these summaries is, however, left for future work.

5.2 EQ2: Performance of Tool Independent Summaries

We measure the performance gains that can be obtained through the use of symbolic summaries implemented with our API and compare the performance of tool-independent summaries against that of native summaries. In order to carry out this experiment we set up a symbolic test suite focusing on LIBC function usage. To the best of our knowledge, no such test suite exists in the literature. In particular, we have analysed both the TestComp [11, 12] and SVComp [10, 13] test suites, counting for each test suite the number of calls to LIBC functions. We concluded that both these test suites make scarce use of LIBC functions, each calling fewer than 3 functions per test on average. Furthermore, the LIBC functions used in these test suites are mostly called with concrete arguments, rendering the use of symbolic summaries pointless.

■ **Table 4** Summarized results in *angr* and *AVD* for both datasets.

			Memory Out	Timeout	Success	Avg.	Avg.	Avg.	Avg.
			×	×	✓	N_{Paths}	N_{Libc}	N_{API}	$Time (s)$
Hash Map	<i>angr</i>	Concrete	7	0	3	2.2k	6.3k	3.6k	–
		C-Summaries	0	0	10	80	419	7.7k	199.66
		Native Summ	0	0	10	72	390	222	97.99
	<i>AVD</i>	Concrete	0	7	3	6.2k	9.6k	1.7k	–
		C-Summaries	0	0	10	95	483	8.4k	61.55
		Native Summ	0	0	10	96	487	244	26.66
Dynamic Strings	<i>angr</i>	Concrete	6	2	4	2.3k	1.0k	4.4k	–
		C-Summaries	1	0	11	431	397	4.0k	235.58
		Native Summ	1	0	11	353	237	97	143.96
	<i>AVD</i>	Concrete	0	7	5	3.5k	5.0k	105	–
		C-Summaries	0	1	11	499	1.9k	1.8k	14.20
		Native Summ	0	1	11	564	1.4k	97	18.94

Experimental Set-up and Symbolic Test Suites

As the test bed for this experiment, we used *angr* [50] and *AVD* [45] extended with our symbolic reflection API (see §3.4). All tests were run on a Ubuntu server (18.04.5 LTS) with an Intel Xeon E5–2620 CPU and 32GB of RAM. Additionally, each test was given 16GB of RAM with a maximum timeout of 30 minutes (1800 seconds).

We searched GitHub for open source C libraries that make intensive use of LIBC string-processing functions. We found two such libraries: (1) the *HashMap* library [55], which provides an implementation of a standard array-based hash table, and (2) the *Dynamic Strings* library [46], which provides an implementation of heap-allocated strings that extend the functionality offered by LIBC strings. Neither of these libraries came with concrete test suites. We created a symbolic test suite for each library: 10 symbolic tests for *HashMap* and 12 symbolic tests for *Dynamic Strings*. The symbolic test suites cover all the functions exposed by two libraries that interact with LIBC functions.

Results

We ran both symbolic test suites on *angr* and *AVD* using: (1) our tool-independent symbolic summaries implemented in C (C Summaries); (2) the native symbolic summaries originally included in each tool (Native Summaries); and (3) the corresponding C reference implementations. Part of these implementations were obtained from *Verifiable C* [4], a toolset for proving the functional correctness of C programs which comes with verified LIBC implementations [3]. The remaining functions were obtained from *glibc* [23] and *libiberty* [22] LIBC implementations.

Results are summarised in Table 4, which shows for each test suite run: (1) the number of tests that failed due to lack of memory (Memory Out); (2) the number of tests that failed for exceeding the time limit (Time Out); (3) the number of tests that finished executing within the given time limit (Success); (4) the average number of explored paths per test (Avg. N_{Paths}); (5) the average number of calls to LIBC functions per test (Avg. N_{Libc}); (6) the average number of calls to API primitives per test (Avg. N_{API}); and (7) the average execution time per test (Avg. $Time$). Note that we do not include the average execution time for the concrete summaries as the majority of the corresponding executions do not finish within the time limit. In contrast, we do include the average execution time for both C summaries and native summaries given that they execute successfully the exact same set of tests.

Unsurprisingly, results clearly show that symbolic summaries substantially improve the performance of symbolic execution tools. For the *HashMap* test suite, we observe that, for both tools, 7 out of the 10 symbolic tests fail to execute without summaries. Using reference implementations, all but the three smallest tests, either exhausted all the available memory or hit the timeout of 30 minutes. The results for the *Dynamic Strings* test suite are analogous. When using reference implementations: 8 out of 12 tests fail to execute with *anqr* and 7 out of 12 tests fail to execute with *AVD*. When it comes to the path explosion problem, we observe that, for both libraries, the number of execution paths generated by symbolic execution with reference implementations is always at least one order of magnitude greater than that generated by symbolic execution with summaries. Table 4 also shows the average number of calls to LIBC functions and API functions. The number of calls to LIBC functions is lower for symbolic execution with summaries than with reference implementations; this is expected as symbolic execution with reference implementations generates a much larger number of paths. Note that, there are calls to the symbolic reflection API even when using reference implementations; this is due to the fact that we always model system calls with summaries.

Finally, we observe that tool-independent summaries are generally less performant than native summaries implemented directly within the code base of the corresponding tools. This slowdown is expected since tool-independent summaries are interpreted, whereas tool-specific summaries are executed natively. Furthermore, our test suites were specifically created to make heavy use of LIBC, making the slowdown more significant than for typical codebases, which interact less frequently with LIBC functions. Importantly, the proposed reflection API was not designed to obtain either performance or expressivity gains with respect to tool-specific summaries, but rather to allow for the implementation of verified, tool-independent summaries that can be shared across multiple SE tools. If the performance of a given tool-independent summary becomes an execution bottleneck for a specific application, then that summary should be implemented natively for the job at hand. However, we believe that will not be the case for the majority of summaries.

5.3 EQ3: Bugs in Symbolic Execution tools

In order to test the applicability of `SUMBOUNDVERIFY`, we used it to find bugs in summaries used in three high-profile symbolic execution tools: *anqr* [50], *Binsec* [19] and *Manticore* [38]. Additionally, we also used `SUMBOUNDVERIFY` to verify the summaries that came with the *AVD* tool. In the following, we classify a summary as *buggy* if it is neither *under-* nor *over-approximating* and if there is no additional information about the expected behaviour of the summary regarding missing/incorrect paths (for instance, in the form of a code comment clarifying how the inputs should be constrained). As we implemented our API in both *AVD* and *anqr*, we were able to use their summaries directly. In contrast, we had to manually re-write *Binsec*'s and *Manticore*'s summaries using our reflection API, following the original algorithms line-by-line. Using this methodology, we were able to validate a total of 52 LIBC symbolic summaries against their corresponding reference implementations.³

The results for all the validated summaries are shown in Table 5. Out of the analysed summaries, we found 14 buggy summaries in *anqr*, 9 in *Binsec*, 1 in *Manticore*, and 13 in *AVD*. These summaries include spurious paths and exclude correct paths, meaning that they are neither under- nor over-approximating. Importantly, only a few summaries included

³ As in §5.2, we use as reference the LIBC implementations from the *Verifiable C* tool chain, and the *glibc* and *libiberty* libraries

24:22 Toward Tool-Independent Summaries for Symbolic Execution

comments restricting the conditions under which they could be soundly applied. However, even these summaries contained bugs that were not ruled out by their authors’ comments, which clearly demonstrates that the development of sound symbolic summaries is a difficult and error prone task that requires automated assistance.

■ **Table 5** Summary bugs found in state-of-the-art symbolic execution tools.

	$N_{Evaluated}$	N_{Bugs}	Bugs Found					
<i>angr</i>	24	14	atoi strncat strtol	atol strncmp strtoul	strcasecmp strncpy wcscasecmp	strchr strstr wcscmp	strcmp strtok_r	
<i>Binsec</i>	9	9	memcmp strcpy	memcpy strncmp	memset strncpy	strchr strchr	strcmp	
<i>Manticore</i>	4	1	strcmp					
Total	37	24						
<i>AVD</i>	15	13	atoi strcat strtol	memcmp strchr tolower	memcpy strcmp toupper	memmove strncmp	memset strncpy	
Total (incl. <i>AVD</i>)	52	37						

To illustrate the type of bug uncovered by SUMBOUNDVERIFY, we present three bugs, each corresponding to a different tool and all three to the function `strcmp`. This function is used to compare two strings lexicographically, returning: (i) a positive integer if the first string is greater than the second one, (ii) a negative integer if it is lower, and (iii) 0 if the two strings coincide. Even though the reference implementation of this function is very compact with less than 10 LoC, its corresponding summaries can be extremely complex (for instance, *angr*’s summary has 160 LoC).

Bug in *angr*

The possible execution paths for the `strcmp` function can be divided into two main sets according to the returned value: the execution paths where the return value is equal to zero and those where it is different. *angr* correctly models all the execution paths that return the value zero, i.e., the cases where the two symbolic input strings are equal. Regarding the execution paths with a return value different from zero, i.e, the cases where the input strings are different, *angr* always constrains the return value to 1. Hence, by returning a fixed positive integer for all paths where the two strings differ, the summary does not satisfy any of the correctness properties. Assuming, for example, two symbolic input strings of size 3, *str1* and *str2*, SUMBOUNDVERIFY produces the following counterexample for *angr*’s summary:

Missing Path: $[str1 = [A, A, A, \backslash 0] \wedge str2 = [B, B, B, \backslash 0] \wedge ret = -1]$

Wrong Path: $[str1 = [A, A, A, \backslash 0] \wedge str2 = [B, B, B, \backslash 0] \wedge ret = 1]$

Essentially, the missing path describes a concrete execution that is not covered by the summary, whereas the wrong path describes a behaviour covered by the summary that is not generated by the execution of the concrete function.

Bug in *Manticore*

Manticore's `strcmp` summary iterates over the two strings to build a nested if-then-else formula over pairs of symbolic bytes. This formula means that if two symbolic bytes are different, then the summary must return the difference of those bytes; if they are equal, the summary must return the value 0 when they are the last two bytes of the strings or continue iterating otherwise. For instance, considering two symbolic input strings: $str1 = [c1, c2, \backslash 0]$ and $str2 = [c3, c4, \backslash 0]$, this summary will create the following if-then-else formula:

$$ret = \text{ITE}(c1 \neq c3, c1 - c3, \text{ITE}(c2 \neq c4, c2 - c4, 0))$$

Even though it appears to be correct, *Manticore*'s summary does not take into account the fact that intermediate symbolic bytes may also be the null character ($\backslash 0$). When validating this summary on the input strings $str1$ and $str2$, `SUMBOUNDVERIFY` produces the following counterexample:

$$\text{Missing Path: } [str1 = [\backslash 0, A, \backslash 0] \wedge str2 = [\backslash 0, B, \backslash 0] \wedge ret = 0]$$

$$\text{Wrong Path: } [str1 = [\backslash 0, B, \backslash 0] \wedge str2 = [\backslash 0, A, \backslash 0] \wedge ret = 1]$$

Bug in *Binsec*

Unlike the previous summary, *Binsec*'s `strcmp` summary takes into account the case where the intermediate symbolic bytes are null characters, but still gets it wrong. For instance, given the same two symbolic input strings: $str1 = [c1, c2, \backslash 0]$ and $str2 = [c3, c4, \backslash 0]$, this summary generates the following formula:

$$\begin{aligned} ret = & \text{ITE}(c1 = \backslash 0, \\ & \text{ITE}(c3 = \backslash 0, 0, 1), \\ & \text{ITE}(c1 = c3, \\ & \quad \text{ITE}(c2 = \backslash 0, \\ & \quad \quad \text{ITE}(c4 = \backslash 0, 0, 1), \\ & \quad \quad \text{ITE}(c2 = c4, 0, \text{ITE}(c2 > c4, 1, -1))), \\ & \text{ITE}(c1 > c3, 1, -1))) \end{aligned}$$

Note that, when comparing each pair of symbolic bytes, this formula first checks if the current byte of $str1$ (e.g., $c1$) is equal to the null character, in which case it then checks if the corresponding byte of $str2$ (e.g., $c3$) is also equal to null; if it is, it evaluates to 0; otherwise, it evaluates to 1 and here lies the problem. When validating this summary on the input strings $str1$ and $str2$, `SUMBOUNDVERIFY` produces the following counterexample:

$$\text{Missing Path: } [str1 = [A, \backslash 0, \backslash 0] \wedge str2 = [A, B, \backslash 0] \wedge ret = -1]$$

$$\text{Wrong Path: } [str1 = [A, \backslash 0, \backslash 0] \wedge str2 = [A, B, \backslash 0] \wedge ret = 1]$$

When the first string is shorter than the second one, *Manticore*'s summary assumes that `strcmp` returns 1 when it should instead return -1 .

Bug in Verifiable C

During our validation experiments we saw unexpected results when using the `strcmp` implementation of *Verifiable C* to validate the corresponding symbolic summaries. In particular, we observed different results for the same `strcmp` summaries when using as reference implementation that of *Verifiable C* and those of the *glibc* and *libiberty* libraries. We found

a bug in the `strcmp` implementation of *Verifiable C* [53]: it compares characters as signed values instead of unsigned ones as mandated by the POSIX specification of `libc`. The code had been proven correct, but for a specification too weak to bring the bug to light. This illustrates yet another application of `SUMBOUNDVERIFY`: it can be used to validate reference implementations against each other.

6 Related Work

There is a vast body of work on summaries for different types of program analysis, such as program logics [30, 44], abstract interpretation [56], and symbolic execution [25, 29]. However, in contrast to program logics, which are typically designed to be compositional and therefore place a heavy emphasis on summaries, in the form of function specifications and their usage, the study of summaries in symbolic execution literature is much more uneven. In particular, while there is a large number of symbolic execution tools that make use of operational summaries in the style of those described in this paper [50, 19, 8, 15, 38, 43, 17, 16, 31], we believe we are the first to address the issue of their formalisation and verification. The existing work on the use of summaries in symbolic execution can broadly be divided into two main groups: (1) *first-order summaries* that either do not reason about the heap memory or do so in a very limited way; and (2) *structural summaries* that rely on various types of representations to model the effects of heap-manipulating functions. In the following, we give a brief outline of research in both categories of summaries, focusing on the work that is closest to ours.

Godefroid et al. were the first to explore the use of first-order summaries in symbolic execution [24, 2, 28]. The first compositional tool in this line of work was `SMART` [24], a dynamic symbolic execution tool with support for summaries. `SMART` analyses functions in isolation in a bottom-up manner, encoding the testing results of each function as a first-order summary to be re-used in the analysis of other functions. Then, the authors proposed a variation of their original algorithm to allow for the lazy exploration of the search space in a top-down manner [2]. Later, the authors presented `SMASH` [28], a framework for testing and verifying C programs. Analogously to `SMART`, `SMASH` is incremental, analysing one function at a time and generating summaries that can be used in the analysis of other functions. The novelty of `SMASH` is that it allows for the combined use of both under- and over-approximating summaries in a demand-driven way. Importantly, the summaries of all three tools consist only of first-order formulas that cannot describe heap effects.

First-order summaries have also been used in the context of loop-summarisation [27, 51, 35]. These works typically combine symbolic execution with a custom-made static analysis component for detecting the induction variables of the loops to be summarised and constructing partial invariants describing their behaviour. Along this line of research, Kapus et al. [32] have recently proposed a new algorithm for inferring loop invariants for string-manipulating C programs using counter-example guided synthesis [1, 49]. These works are, however, complementary to ours since our goal is not to automatically generate summaries but rather to validate manually-written ones.

When it comes to structural summaries, Qiu et al. [42] proposed a new approach for compositional symbolic execution where function summaries are expressed as *memoization trees*. A *memoization tree* is a tree-like data structure that describes the various paths taken by the summarised function, *including* their effects on the heap. To this end, memoization tree nodes describe both the variable store as well as the contents of the heap of the summarised function. The proposed tool is compositional, analysing one function at a time and expanding the contents of symbolic objects by need, following the lazy initialisation approach [33].

Recently, Frago Santos et al. [48] proposed JaVerT 2.0, a new compositional symbolic execution tool for JavaScript. JaVerT 2.0 combines separation-logic-based summaries with static symbolic execution. More concretely, it allows for the incremental analysis of the given program, generating separation-logic-based specifications that can later be used during symbolic execution. The use of separation-logic summaries during symbolic execution has also been explored in the design of the Gillian framework [21, 36], which resulted from the generalisation of JaVerT 2.0 to a multi-language setting.

7 Conclusions

Symbolic summaries are a key element of modern symbolic execution engines. They are an essential tool for both containing the path explosion problem and modelling interactions with the runtime environment. The implementation of symbolic summaries is time-consuming and error-prone, but until now there was a lack of mechanisms and methodologies for sharing symbolic summaries across different tools and for their validation.

This paper proposes a new API for developing and verifying tool-independent summaries. Using the proposed API, symbolic summaries can be directly implemented in C and shared across different symbolic execution tools, provided that these tools implement the API. To demonstrate the expressiveness of our API, we extended the symbolic execution tools *anqr* and *AVD* with support for it and developed tool-independent symbolic summaries for 26 different LIBC functions, comprising string-manipulating functions, number-parsing functions, input/output functions, and heap functions. Furthermore, we presented *SUMBOUNDVERIFY*, a new tool for the bounded verification of the summaries written with our symbolic reflection API. We applied *SUMBOUNDVERIFY* to 37 symbolic summaries taken from 3 state-of-the-art symbolic execution tools, *anqr*, *Binsec* and *Manticore*, detecting a total of 24 buggy summaries.

As future work, we intend to design a tool for automating the creation of symbolic summaries by synthesising them from declarative specifications, such as separation logic triples. To this end, we plan to leverage recent results on code synthesis from separation logic specifications [41], with the key difference being that our goal is to synthesise symbolic summaries instead of reference implementations.

References

- 1 Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 270–288, Cham, 2018. Springer International Publishing.
- 2 Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–381, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 3 Andrew Appel. The Verifiable C string library, 2021. Software distribution that accompanies [4]. URL: <https://softwarefoundations.cis.upenn.edu/vc-current/index.html>.
- 4 Andrew W. Appel, Lennart Beringer, and Qinxiang Cao. *Verifiable C*, volume 5 of *Software Foundations*. Electronic textbook, 2021. URL: <http://softwarefoundations.cis.upenn.edu>.
- 5 Krzysztof R. Apt. Ten Years of Hoare’s Logic: A Survey—Part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, October 1981. doi:10.1145/357146.357150.
- 6 Roberto Baldoni, Emilaio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018. doi:10.1145/3182657.

- 7 Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods*, pages 1–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- 8 Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis*, pages 201–207, Cham, 2017. Springer International Publishing.
- 9 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification CAV 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
- 10 Dirk Beyer. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–422, Cham, 2021. Springer International Publishing.
- 11 Dirk Beyer. Status Report on Software Testing: Test-Comp 2021. In *Fundamental Approaches to Software Engineering*, pages 341–357, Cham, 2021. Springer International Publishing.
- 12 Dirk Beyer. Advances in Automatic Software Testing: Test-Comp 2022. In *Fundamental Approaches to Software Engineering - 25th International Conference, FASE 2022*, volume 13241 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2022.
- 13 Dirk Beyer. Progress on software verification: SV-COMP 2022. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*, volume 13244 of *Lecture Notes in Computer Science*, pages 375–402. Springer, 2022.
- 14 Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Not.*, 10(6):234–245, April 1975. doi:10.1145/390016.808445.
- 15 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI’08*, pages 209–224, USA, 2008. USENIX Association.
- 16 Marek Chalupa, Vincent Mihalkovič, Anna Řečtáčková, Lukáš Zaoral, and Jan Strejček. Symbiotic 9: String analysis and backward symbolic execution with loop folding. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 462–467, Cham, 2022. Springer International Publishing.
- 17 Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems - TOCS*, 30:1–49, February 2012. doi:10.1145/2110356.2110358.
- 18 L. Daniel, S. Bardin, and T. Rezk. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038, Los Alamitos, CA, USA, May 2020. IEEE Computer Society. doi:10.1109/SP40000.2020.00074.
- 19 R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656, 2016. doi:10.1109/SANER.2016.43.
- 20 Leonardo de Moura and Nikolaž Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- 21 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, Part I: A multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*. Association for Computing Machinery, 2020.

- 22 GNU. GNU libiberty, 2022. Accessed: 6th July 2023. URL: <https://gcc.gnu.org/onlinedocs/libiberty/>.
- 23 GNU. The GNU C library, 2022. Accessed: 6th July 2023. URL: <https://www.gnu.org/software/libc/>.
- 24 Patrice Godefroid. Compositional Dynamic Test Generation. *SIGPLAN Not.*, 42(1):47–54, January 2007. doi:10.1145/1190215.1190226.
- 25 Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 112–128. Springer, 2011. doi:10.1007/978-3-642-23702-7_12.
- 26 Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.
- 27 Patrice Godefroid and Daniel Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11*, pages 23–33, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2001420.2001424.
- 28 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 43–56. ACM, 2010. doi:10.1145/1706299.1706307.
- 29 Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *Computer Aided Verification*, pages 68–81, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- 30 Samin S. Ishtiaq and Peter W. O’Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK*, pages 14–26. ACM, 2001. doi:10.1145/360204.375719.
- 31 Joxan Jaffar, Rasool Maghareh, Sangharatna Godbole, and Xuan-Linh Ha. TracerX: Dynamic Symbolic Execution with Interpolation (Competition Contribution). In *Fundamental Approaches to Software Engineering*, pages 530–534, Cham, 2020. Springer International Publishing.
- 32 Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. Computing summaries of string loops in c for better testing and refactoring. In *PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 874–888, New York, NY, USA, 2019. Association for Computing Machinery.
- 33 Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003*, volume 2619 of *Lecture Notes in Computer Science*. Springer, 2003.
- 34 James C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233, New York, NY, USA, 1975. Association for Computing Machinery. doi:10.1145/800027.808444.
- 35 Yude Lin, Tim Miller, and Harald Søndergaard. Compositional symbolic execution using fine-grained summaries. In *2015 24th Australasian Software Engineering Conference*, pages 213–222, 2015. doi:10.1109/ASWEC.2015.32.
- 36 Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: real-world verification for javascript and C. In *Computer Aided Verification - 33rd International Conference, CAV 2021*, volume 12760 of *Lecture Notes in Computer Science*, pages 827–850. Springer, 2021.

- 37 Petar Maksimovic, Caroline Cronjäger, Julian Sutherland, Andreas Lööw, Sacha-Élie Ayoun, and Philippa Gardner. Exact separation logic. *CoRR*, abs/2208.07200, 2022. [arXiv:2208.07200](https://arxiv.org/abs/2208.07200).
- 38 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019. [doi:10.1109/ASE.2019.00133](https://doi.org/10.1109/ASE.2019.00133).
- 39 Manh-Dung Nguyen, S'ebastien Bardin, Richard Bonichon, R. Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. *ArXiv*, abs/2002.10751, 2020. [arXiv:2002.10751](https://arxiv.org/abs/2002.10751).
- 40 Peter O'Hearn. Incorrectness logic. *Proceedings of the ACM on Programming Languages*, 4:1–32, December 2019. [doi:10.1145/3371078](https://doi.org/10.1145/3371078).
- 41 Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL):72:1–72:30, 2019.
- 42 Rui Qiu, Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. Compositional symbolic execution with memoized replay. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 632–642, 2015. [doi:10.1109/ICSE.2015.79](https://doi.org/10.1109/ICSE.2015.79).
- 43 E. Reisner, C. Song, K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 445–454, 2010. [doi:10.1145/1806799.1806864](https://doi.org/10.1145/1806799.1806864).
- 44 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science LICS 2002*, pages 55–74. IEEE Computer Society, 2002.
- 45 Nuno Sabino. Automatic vulnerability detection: Using compressed execution traces to guide symbolic execution. Master's thesis, Instituto Superior Técnico, November 2019.
- 46 Salvatore Sanfilippo. Simple dynamic strings, 2015. Accessed: 6th July 2023. URL: <https://github.com/antirez/sds>.
- 47 José Fragoso Santos, Petar Maksimovic, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic Execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, pages 11:1–11:14. ACM, 2018.
- 48 José Fragoso Santos, Petar Maksimovic, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proc. ACM Program. Lang.*, 3(POPL):66:1–66:31, 2019.
- 49 Vaibhav Sharma, Kesha Hietala, and Stephen McCamant. Finding substitutable binary code by synthesizing adapters. *IEEE Transactions on Software Engineering*, PP:1–1, July 2019. [doi:10.1109/TSE.2019.2931000](https://doi.org/10.1109/TSE.2019.2931000).
- 50 Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016. [doi:10.1109/SP.2016.17](https://doi.org/10.1109/SP.2016.17).
- 51 Jan Strejček and Marek Trtík. Abstracting Path Conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*, pages 155–165, New York, NY, USA, 2012. Association for Computing Machinery. [doi:10.1145/2338965.2336772](https://doi.org/10.1145/2338965.2336772).
- 52 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013*, pages 135–152, New York, NY, USA, 2013. Association for Computing Machinery. [doi:10.1145/2509578.2509586](https://doi.org/10.1145/2509578.2509586).
- 53 Verifiable C – Verif_strlib. Bug in strcmp function. Accessed: 6th July 2023. URL: https://softwarefoundations.cis.upenn.edu/vc-current/Verif_strlib.html.

- 54 Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10:203–232, April 2003. doi:10.1023/A:1022920129859.
- 55 Richard Wiedenhöft. C Hash map, 2014. Accessed: 6th July 2023. URL: <https://gist.github.com/Richard-W/9568649>.
- 56 Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 221–234. ACM, 2008.

A Direct-Style Effect Notation for Sequential and Parallel Programs

David Richter ✉ 

Technische Universität Darmstadt, Germany

Timon Böhler ✉ 

Technische Universität Darmstadt, Germany

Pascal Weisenburger ✉ 

Universität St. Gallen, Switzerland

Mira Mezini ✉ 

Technische Universität Darmstadt, Germany

hessian.AI, Darmstadt, Germany

Abstract

Modeling sequential and parallel composition of effectful computations has been investigated in a variety of languages for a long time. In particular, the popular `do`-notation provides a lightweight effect embedding for any instance of a monad. Idiom bracket notation, on the other hand, provides an embedding for applicatives. First, while monads force effects to be executed sequentially, ignoring potential for parallelism, applicatives do not support sequential effects. Composing sequential with parallel effects remains an open problem. This is even more of an issue as real programs consist of a combination of both sequential and parallel segments. Second, common notations do not support invoking effects in direct-style, instead forcing a rigid structure upon the code.

In this paper, we propose a mixed applicative/monadic notation that retains parallelism where possible, but allows sequentiality where necessary. We leverage a direct-style notation where sequentiality or parallelism is derived from the structure of the code. We provide a mechanisation of our effectful language in Coq and prove that our compilation approach retains the parallelism of the source program.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Software and its engineering → Concurrent programming structures; Software and its engineering → Parallel programming languages

Keywords and phrases `do`-notation, parallelism, concurrency, effects

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.25

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.17>

Software: <https://github.com/stg-tud/parseq-notation>

archived at `swh:1:dir:b77c1bb5bff47b79976c4134d1cb143c2a7b8704`

Funding *David Richter:* German Federal Ministry of Education and Research *iBlockchain project* (BMBF No. 16KIS0902).

Timon Böhler: Hessian Ministry of Higher Education, Research, Science and the Arts via the project 3rd Wave of AI (3AI).

Pascal Weisenburger: The University of St. Gallen (IPF, No. 1031569); Swiss National Science Foundation (SNSF, No. 200429).

Mira Mezini: Hessian Ministry of Higher Education, Research, Science and the Arts via the project 3rd Wave of AI (3AI); BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*; German Federal Ministry of Education and Research *iBlockchain project* (BMBF No. 16KIS0902).



© David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 25; pp. 25:1–25:22



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Programming language designers often select a few common effects (state, IO, network) and bake them into the language. It is, however, impossible to predict what effects developers will need in the future (as was the case with integrated queries [32, 30, 4], reactive programming [31], asynchronous programming [51, 5], multitier programming [36, 43], differentiable programming [23], etc). Thus, we argue that language designs should be equipped with support for developer-implementable effects.

Modeling effectful computations has long been the subject of investigation in the context of various languages. As a result, there exist different abstractions and notations with different properties. A prominent abstraction for modeling effectful computations are monads (e.g., known from Haskell) and the do-notation that emerged from monadic comprehensions [55] providing a lightweight way to embed monads into programs. But monads force effects to be executed sequentially, ignoring potential for parallelism. Notations for parallelism, such as idiom bracket notation for applicatives [29], on the other hand, do not support sequential effects. Yet, programs are rarely only parallel or only sequential; thus it is desirable to support both sequential and parallel composition of effectful operations.

To the best of our knowledge, there are no approaches that provide such support. The `ApplicativeDo` approach by Marlow et al. [27, 28] attempts to retrofit parallelism into the do-notation, i.e., with `ApplicativeDo`, developers write code using the do-notation and an optimising compiler tries to infer which computations are parallelizable. Yet, in the general case, it is not possible to decide statically whether two monadic operations are actually parallelizable or whether the result of one operation depends on the execution of the other. Hence, there is a danger that the compiler either incorrectly decides that two operations can be executed in parallel, which can lead to race conditions, or conservatively decides to not parallelize operations that could actually be parallelized, reducing the potential for improved performance. To counteract race conditions, the `ApplicativeDo` approach requires developers to adhere to specific coding conventions such as only using expressions which are either all read-only or write-only [27].

Another weak point of Haskell's do-notation (and thus also of the approach by Marlow et al.) is that it enforces a specific structure upon the code with strict adherence to one effect per line, which does not allow effects in arbitrary places. The do-notations in Idris [21] and Lean [25, 52] are less restrictive and support direct-style effect usage. Scala supports both structuring effectful code in do-notation via for-comprehensions and for in some cases in direct-style via `async-await` [44]; but, both are based on monads, thus force sequential execution. Although `async-await` was explicitly designed for concurrency, developers must be careful to start parallel execution before accessing their result to preserve parallelism.

In this paper, we propose a direct-style notation that enables sequential and parallel composition of effectful operations (using monads and applicatives, respectively) without forcing a specific structure of the code. We present a one-pass translation from direct-style to monadic effect combinators. Instead of trying to infer the potential for parallelism on top of sequential programs, our approach preserves parallelism that is inherent in the structure of the code thanks to direct style. This makes it easier to reason about the correctness of the proposed translation process and we present a correctness proof and its mechanization in Coq [9]. We conceptualize the preservation of parallelism as the span of a term (the length of the longest path of effectful operations) and the work of a term (the sum of all effectful operations therein). Our translation is span-preserving leveraging applicatives and monads. In contrast, notations based on monads alone are not span-preserving, as they have to chain all effectful operations into a single sequence.

Our compilation has an elegant description as a set of equations forming a structurally recursive function over the syntax, whose equations are the well-known monadic and applicative laws and free theorems. Implementations for do-notation are essentially compilers for an effectful language. They can produce efficient code by avoiding administrative redexes and generating proper tail calls. Including this optimisation in standard effectful languages modeled by monads can be seen as performing partial evaluation of the code via the semantics, extended by the monad laws. In our case, we target mixed monadic and applicative code. Therefore, our optimised translation also combines the use of the monadic laws with the use of applicative laws.

Contributions. In summary, this paper makes the following contributions:

- We present the first mixed applicative/monadic direct-style effect notation.
- We formalize an optimised one-pass translation from direct-style to effect combinators.
- We prove that our translation preserves typability, semantics, and parallelism.
- We mechanize the proof using parametric higher order abstract syntax.
- We implement the proposed translation in the Scala programming language using Scala macros, which enables us to stay close to the formal development.

Structure. The remainder of the paper is structured as follows. Section 2 provides code examples and an intuitive overview of our approach. Section 3 formally defines the proposed translation and provides a proof that it preserves typability, semantics, and parallelism, which is mechanized using parametric higher order abstract syntax. Section 4 presents the implementation in Scala. Section 5 surveys related work. Section 6 concludes the paper and presents ideas for future work.

2 Overview

In this section, we (a) briefly discuss the difference between monadic, applicative, and mixed notations by examples in Scala, and (b) informally present our mixed direct notation and its implementation by translation to effect combinators.

2.1 Monadic, Applicative, Mixed and Direct Style Notations

Functors, Applicatives and Monads. A functor (in functional programming) for $F: \text{Type} \rightarrow \text{Type}$ is a method `map` that turns a function on values into a function on values wrapped in the functor. Intuitively, a value of type $F\ A$ represents an effectful computation of type A . An applicative for F is a functor and a method `pure` to wrap a value into the functor, and a method `ap` (which we occasionally also write $f \diamond \times$ instead of `ap f x`) to apply an effectful computation returning a function, to an effectful computation returning an argument. A monad for F is an applicative for F and a method `bind`, which runs an effectful computation and feeds the resulting value to another effectful computation. Below we show the mathematical description and an encoding in Scala via traits:

<pre>map: (A → B) → (F A → F B) pure: A → F A ap: F (A → B) → (F A → F B) bind: (A → F B) → (F A → F B)</pre>	<hr/> <pre>trait Functor[F[_]]: def map(f: A ⇒ B, a: F[A]): F[B] trait Applicative[F[_]] extends Functor[F]: def pure(a: A): F[A] def ap(f: F[A ⇒ B], a: F[A]): F[B] trait Monad[F[_]] extends Applicative[F]: def bind(f: A ⇒ F[B], a: F[A]): F[B]</pre> <hr/> <p style="text-align: right; font-size: small;">Scala</p>
---	---

25:4 A Direct-Style Effect Notation for Sequential and Parallel Programs

For convenience, we will write `pure(x)`, `x.bind(f)` and `f.ap(x)`, so we define them in Scala as a `extension` methods for every object which has a corresponding instance, and `pure(x)` as a top-level function. Note that we swapped arguments for `bind` as a method `x.bind(f)` with regard to its type as a function `bind(f)(x)`.

Monadic Notation. To illustrate monadic notations, consider the two lines of code below (left side) that use a for-comprehension `for ... yield ...`¹. The programs execute two effectful statements `fetchX` and `fetchY` and bind the result in variables `x` and `y`, respectively, to be combined by a function call to `f`. The for-comprehension (monadic notation) is desugared into explicit use of monadic `bind` (right side).

<pre style="margin: 0;">for x ← fetchX; y ← fetchY yield f(x)(y) for y ← fetchY; x ← fetchX yield f(x)(y)</pre>	<pre style="margin: 0;">fetchX.bind(x ⇒ fetchY.bind(y ⇒ pure(f(x)(y)))) fetchY.bind(y ⇒ fetchX.bind(x ⇒ pure(f(x)(y))))</pre>
Scala	Scala

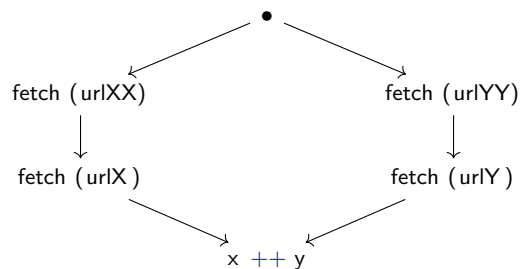
Applicative Notation. In the programs above, `x` does not depend on `y` and vice versa. If all statements in the program of an applicative are independent from each other – e.g., none of the variables that are introduced in the for-comprehension are used in the `for` part, but only after the `yield`, which has access to all variables introduced above – we can interpret the for-comprehension as an applicative notation instead of monadic notation. Then, the program below on the left side would be translated into the program below on the right side, using applicative `ap` to encode actual parallelism, where `ap` is parallel execution followed by function application. In the example program, `pure(f)`, `x` and `y` are executed in parallel, followed by the function application of the result of `pure(f)` to the result of `x` and the result of `y`:

<pre style="margin: 0;">for x ← fetchX; y ← fetchY yield f(x)(y)</pre>	<pre style="margin: 0;">pure(f).ap(fetchX).ap(fetchY)</pre>
Scala	Scala

Mixed Notation. To illustrate where these notations for effectful computations fall short, consider the following larger program that fetches four resources from the Internet. The program first fetches a resource `urlXX`, which contains another url `urlX`, and then fetches a resource from `urlX` and stores it in `x`. The program further fetches a resource `urlYY`, which contains another url `urlY`, and then fetches the resource from `urlY` and stores it in `y`. Finally, the program concatenates `x` and `y`:

```
val urlXX = "https:// example.org/configx "
val urlYY = "https:// example.org/configy "
for urlX ← fetch(urlXX)
  x ← fetch(urlX)
  urlY ← fetch(urlYY)
  y ← fetch(urlY)
yield x ++ y
```

Scala



Observe that `urlXX` needs to be fetched before `urlX` can be fetched and `urlYY` needs to be fetched before `urlY` can be fetched. But fetching `urlXX` and `urlYY` is independent from each other; and so is `urlX` and `urlY` (as illustrated in the diagram above). Thus, the example contains

¹ For-comprehensions `for ... yield ...` are Scala's equivalent of Haskell's do-notation `do ...; return ...`. The main difference besides Scala's and Haskell's monadic notation is that every for-comprehension must end with a `yield`. Yet, this does not reduce expressive power, as any do-block without a final return can be expressed with an additional binding, e.g., `do ...; e` can be represented by `for ...; tmp ← e yield tmp`.

both parallel and sequential elements. However, if implemented via monadic notation, the program will run sequentially, fetching `urlXX`, then `urlX`, then `urlYY`, then `urlY`. The applicative notation, on the other hand, is not even possible because it would require all effects to be independent from each other.

Direct-Style Mixed Notation. In our direct-style notation, we combine the syntactic form `for ... yield ...` into a single instruction `purify`. Now, the program can be written to look like the following snippet, which reads *Concatenate (1) the result of fetching the value pointed to by a URL by fetching `urlXX` with (2) the result of fetching the value pointed to by a URL by fetching `urlYY`*:

```
purify:
  fetch(fetch(urlXX).↓).↓ ++ fetch(fetch(urlYY).↓).↓
```

Scala

The `purify` operation introduces an operation of type $\downarrow: F[X] \Rightarrow X$ used like `... .↓` into the local scope, which represents direct-style effect execution. If the enclosed code does not make use of the `↓`, the operation `purify` works exactly like `pure`. Otherwise, effect execution `↓` is translated into proper use of `bind` and `ap`.

In direct style, potential for parallelism is implicitly defined by the structure of the code. In particular, the function arguments of `concat ++` naturally have no dependency on each other, and can therefore be executed in parallel. The corresponding program with explicit mixed monadic/applicative combinators is:

```
pure(x => y => x ++ y).ap( fetch(urlXX).bind(fetch) ).ap( fetch(urlYY).bind(fetch) )
```

Scala

The approach we propose in this paper exploits the information encoded in our direct-style notation to define a compositional (e.g., structurally recursive) and provably correct compiler that transforms such direct-style programs into semantically equivalent mixed applicative/monadic programs.

2.2 Discussion

We discuss the similarities and differences between different notations. For illustration, consider that monadic code in the `for/yield` notation can be easily refactored into direct style, roughly by replacing `←` with `= ↓`. In turn, monadic code is compiled into explicit use of monadic operators by calling `bind` on each value and calling `map` on the last:

<pre>for x ← a y ← b z ← c yield e</pre> <p style="text-align: right;">Scala</p>	<pre>purify : val x = a.↓ val y = b.↓ val z = c.↓ e</pre> <p style="text-align: right;">Scala</p>	<pre>a.bind { x => b.bind { y => c.map { z => e }}}}</pre> <p style="text-align: right;">Scala</p>
--	---	---

Scaling. A benefit of direct-style code is that it “scales” better for larger programs in the sense that it integrates well with common language constructs. In particular, direct style composes better with branching. Consider the following versions of the same program. On the left, the program is written in monadic notation, implemented in pure Scala, which requires us to leave and enter monadic notation a second time. The program fetches the time of the last change and the last caching of a certain request. If the resource has been updated since the last caching, we count the cache miss for statistics, request the resource freshly,

25:6 A Direct-Style Effect Notation for Sequential and Parallel Programs

pass it to the cache and return it. Otherwise, we count the cache hit for statistics, and return the answer from the cache. Now, compare the program on the left with a clearer direct-style representation on the right, which is implemented using our approach (code on the right).

<pre> for freshtime ← fetch (freshtimeUrl) cachetime ← fetch (cachetimeUrl) result ← if freshtime > cachetime then for _ ← countFresh tmp1 ← fetchFresh tmp2 ← writeCache (tmp1) yield tmp2 else for _ ← countCache tmp ← readCache yield tmp yield result </pre>	<pre> purify : if fetch (freshtimeUrl).↓ > fetch (cachetimeUrl).↓ then countFresh .↓ parseAndCache(fetchFresh .↓).↓ else countCache.↓ readCache.↓ </pre>
Scala	Scala

Sub-notations. Direct-style notation subsumes three different explicit effect notations (Table 1). First, if a `purify` expression contains exactly one down arrow \downarrow as a mark for effect execution (Row 1), then the notation translates to solely using the Functor interface. This case corresponds to a standard map operation.

Second, if a `purify` expression contains multiple such marks, which are “parallel” with regard to each other (Row 2) – i.e., when they are side-by-side inside different arguments to a function – then the expression translates to solely using the Applicative interface. Crucially, in this case, different effect executions cannot depend on each other. We call these effect executions “parallel” as contrasted with “sequential” code, where a statement can depend on the previous one. Such parallel composition enables parallel execution of effects at run time.

Third, if the expression makes use of nested marks (Row 3) – or equivalently of marks which make use of previously bound variables which contained marks (Row 4) – then the expression translates to using the full Monad interface, which models sequential code.

Direct-style enables to define parallelism naturally by structuring code such that the execution of effects are independent, which gives rise to parallel execution of code where this is possible and using sequential execution where necessary.

2.3 From Laws to a Rewrite System

We refresh the laws of functors, applicatives, and monads and give an intuition how they can be used to optimise effectful programs. Then we state a completion of the laws into a rewrite system. We use the symbol \circ for function composition as in $f \circ g$, and use the symbol (\circ) as the name of function composition, when not used as an infix operator, i.e., $(\circ) f g v := f (g v)$.

■ **Table 1** Subnotations.

Scheme	Description	Typeclass
<code>purify{ ... ↓ ... }</code>	one mark	Functor <code>map</code>
<code>purify{ ... ↓ ... ↓ ... }</code>	multiple marks	Applicative <code>ap</code>
<code>purify{ ... (... ...↓ ...)↓ ... }</code>	nested marks	Monad <code>bind</code>
<code>purify{ ...; val x = ...↓; ... ↓ ... }</code>	consecutive marks	Monad <code>bind</code>

Laws. Typically, the coherence laws are formulated as follows [29, 26].

For Functors, `map` preserves identity and composition, i.e., applying the identity function to an effectful computation is the same as not doing anything; and applying two function in sequence to an effectful value is the same as applying the composite once.

```
identity      : map id v      = id v
composition   : map (f ∘ g) v = map f (map g v)
```

For Applicatives, `pure` creates a effect-free, i.e., *pure* value from a value. The *homomorphism law* states that, applying a pure function to a pure argument is pure. The *identity law* states that, if the function is just pure, then there is nothing to do. The *interchange law* states that, if the argument is pure, then we can swap the pure argument with the effectful function.

```
homomorphism  : pure f ∘ pure v = pure (f v)
identity      : pure id ∘ v      = v
interchange   : f ∘ pure v      = pure (λ f', f' v) ∘ f
composition   : u ∘ (v ∘ w)     = pure (∘) ∘ u ∘ v ∘ w
```

For Monads, the first and second laws state that executing a pure computation amounts to not having to execute any effect at all, allowing us to eliminate the `bind`. The third law states that `bind` is associative, i.e., applying two effectful functions in sequence is the same as applying the effectful composite of the two functions once.

```
leftunit     : bind f (pure v) = f v
rightunit    : bind pure v     = v
associativity : bind g (bind f v) = bind (bind g ∘ f) v
```

Free theorems. The laws are phrased as an equational theory – to create a compiler from the laws, we need to rephrase them as a terminating rewrite system. To do so, we first complete our set of equations with the following free theorems. Free theorems hold in programming languages with parametric polymorphism by parametricity for free [54, 53], therefore they are often not stated specifically in the laws, because there is no additional effort required to make them hold. On the other hand, as we are interested in making use of laws for optimisation purposes, we are allowed to make use of the free theorems as well.

Consider the function `pure`: $\forall A, A \rightarrow F A$. Because it must work over all A it cannot change or create new elements of type A , but only duplicate or forget values of type A . Therefore, it does not matter whether we apply a function `g` to change the A s into B s before or after applying `pure`. On the left we apply `f` on the argument of `pure`, on the right we apply `f` on the result:

```
free_pure : map f (pure v) = pure (f v)
```

Similarly, consider the function `ap`: $\forall A B, F (A \rightarrow B) \rightarrow F A \rightarrow F B$ and `bind`: $\forall A B, (A \rightarrow F B) \rightarrow (F A \rightarrow F B)$. On the left we apply `f` on the argument of `ap` and `bind`, on the right we apply `f` on the result, where $(f \circ)$ stands for $\lambda g, f \circ g$:

```
free_ap   : ap (map (f ∘) g) v = map f (ap g v)
free_bind : bind (map f ∘ g) v = map f (bind g v)
```

25:8 A Direct-Style Effect Notation for Sequential and Parallel Programs

We instantiate `g` with `pure id` in the applicative case and with `pure` in the monadic case, then we can extend the equation chain to the left by the free theorem of `pure` (`map f ∘ pure = pure ∘ f`), and to the right by the identity applicative law (`ap (pure id) v = v`) respectively the left-unit monad law (`bind pure v = v`):

```
*free_ap  : ap (pure f) v      = ap (map (f ∘) (pure id)) v = map f (ap (pure id) v) = map f v
*free_bind : bind (pure ∘ f) v = bind (map f ∘ pure) v    = map f (bind pure v)    = map f v
```

In fact, these two equations share a common right-hand side, and thus we can combine them to get a connection between applicative `ap` and monadic `bind`:

```
ap_bind:  ap (pure f) v = bind (pure ∘ f) v
```

Completion. We can use the free theorems to complete the functor, applicative and monad laws into a more suitable form. In particular, we replace the identity law of the applicative with their generalization derived above. Similarly, the right hand side of the interchange law contained the left hand side of the identity law, therefore we simplify it by composition with the identity law. Further, observe that the homomorphism law becomes superfluous, as it can be constructed by applying the identity law (or equivalently by the interchange law) followed by the free theorem of `pure`; however we will still make use of it in swapped direction, such that reading the laws from left to right, it does not overlap with the other applicative laws. The complete set of equations is now:

```
identity      : map id v      = v
composition   : map f (map g v) = map (f ∘ g) v

homomorphism  : pure (f v)    = pure f ∘ pure v          -- swapped
identity      : pure f ∘ v    = bind (λ v', pure (f v')) v -- generalised by ap_bind
interchange   : f ∘ pure v    = bind (λ f', pure (f' v)) f  -- combined with identity
composition   : u ∘ (v ∘ w)   = map (∘) u ∘ v ∘ w          -- combined with identity

leftunit      : bind f (pure v) = f v
rightunit     : bind pure v     = v
associativity  : bind g (bind f v) = bind (bind g ∘ f) v
```

Looking at the equations above, we see that the identity and interchange law show that a non-pure argument to `ap` on the left and on the right can each be represented by a `bind`, so one might think both laws can be unified by a single law, using two binds like `fs ∘ xs = bind (λ f, bind (λ x, pure (f x)) xs) fs`. However, it is not valid to assume this equation. Actually, there are *at least two* possible trivial instances of an applicative for any monad, the left-to-right applicative above, but also the right-to-left applicative: `fs ∘ xs = bind (λ x, bind (λ f, pure (f x)) fs) xs`. There is no reason to prefer one over the other, and, in general, the assumption of either of these equations is too strong; committing to one such equation would allow elimination of all `aps` into `binds`, and thus implies full sequentiality. To support parallelism, we have to make neither assumption and only rely on the equations derived from the applicative laws.

2.4 Translation

We present a rewrite system based on the laws, and prove its terminating by phrasing it as a structurally recursive function.

We distinguish between a source language and a target language below. The source language consists of term formers for variables, function application, and the direct-style effect execution \downarrow represented as `Each`. The target language consists of term formers for variables, function application, and effect combinators `Pure`, `Bind` and `Ap`; and parallelism is explicitly structured by those combinators.

$$\begin{aligned} (\text{Src}) \quad e, f &::= \text{Var } x \mid \text{App } f \ e \mid \text{Each } e \\ (\text{Tgt}) \quad g, h &::= \text{Var } x \mid \text{App } g \ h \mid \text{Pure } g \mid \text{Ap } g \ h \mid \text{Bind } g \ h \end{aligned}$$

The source language uses direct style. In programs written in the source language, parallelism is implicitly defined by the structure of the code. In particular, function arguments naturally have no dependency on each other, and can therefore be executed in parallel. Our compiler translates direct style into monadic and applicative combinators. The essence of our compilation strategy is to use the monadic and applicative laws directly as the actual transformation rules.

Basic Translation. The translation starts with the `PURE` expression, which is implemented as a structurally recursive function over syntax, expanding the direct-style use of effects \leftarrow into the effect operation `Bind`, while variables are wrapped in a `PURE`, and function application is translated to applicative `Ap`, realising that function arguments can be executed in parallel.

$$\begin{aligned} \text{PURE: Src} &\rightarrow \text{Tgt} \\ \text{PURE (Var } x) &= \text{Pure (Var } x) \\ \text{PURE (Each } e) &= \text{Bind (PURE } e) \text{ id} \\ \text{PURE (App } f \ e) &= \text{Ap (PURE } f) \text{ (PURE } e) \end{aligned}$$

Optimising Translation. If we only cared about a correct translation from the direct-style notation to the pure calculus with explicit combinators, then the translation we discussed so far is sufficient. Yet, we consider an optimising translation (Figure 1), where instead of term using the constructors `Bind` and `Ap` (capitalized) directly, we use the smart constructors `AP` and `BIND` (all capitals) instead. Both the constructor and the smart constructor of a term do construct terms that are semantically indistinguishable, i.e., $\text{AP } f \ x \approx \text{Ap } f \ x$. Smart constructors, however, internalize the optimisation by reducing to a simpler term if possible. The translation `PURE` we have described earlier can be seen as such a smart constructor for the `Pure` term constructor. It also preserves the semantics, i.e., $\text{PURE } x \approx \text{Pure } x$.

The only difference between the basic and the optimised translation, is that the optimised smart constructor `PURE` calls to the smart constructor `AP` instead of using the term constructor `Ap` directly, which can lead to further optimisations. In this way, we can leverage smart constructors to integrate the translation with an optimisation into a one-pass optimised translation. For the optimisation, the smart constructors apply the monadic and applicative laws, only in the other direction than the translation, i.e., bubbling up `Pure` in a structurally recursive way through the term, and thereby removing superfluous effect combinators in the generated code.

In particular, `AP` (Figure 1) will reduce the applicative application of a pure function to a pure argument back into the pure function application with only the result wrapped into `Pure` (which is simply the reverse rule of the homomorphism law we used above). Similarly, if either side of `AP` is pure, there are no two effects to be executed in parallel but just a single effect. Hence, we can reduce the term to a single monadic bind. Finally, if neither argument to `AP` is marked as pure, then we simply return the actual term former `Ap` and retain the

25:10 A Direct-Style Effect Notation for Sequential and Parallel Programs

(Src) $e, f ::= \text{Var } x \mid \text{App } f \ e \mid \text{Each } e$		PURE: $\text{Src} \rightarrow \text{Tgt}$																																	
(Tgt) $g, h ::= \text{Var } x \mid \text{App } g \ h$		AP: $\text{Tgt} \rightarrow \text{Tgt} \rightarrow \text{Tgt}$																																	
	$\mid \text{Pure } e \mid \text{Ap } g \ h \mid \text{Bind } g \ h$	BIND: $\text{Tgt} \rightarrow \text{Tgt} \rightarrow \text{Tgt}$																																	
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px 2px 10px;">PURE (Var x)</td> <td style="padding: 2px 10px 2px 10px;">= Pure (Var x)</td> <td style="padding: 2px 10px 2px 10px;">-- <i>indistinguishable</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">PURE (Each e)</td> <td style="padding: 2px 10px 2px 10px;">= BIND id (PURE e)</td> <td style="padding: 2px 10px 2px 10px;">-- <i>effect translation</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">PURE (App $f \ e$)</td> <td style="padding: 2px 10px 2px 10px;">= AP (PURE f) (PURE e)</td> <td style="padding: 2px 10px 2px 10px;">-- <i>homomorphism law</i></td> </tr> <tr> <td colspan="3" style="padding: 10px 0 10px 20px;"> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px 2px 10px;">AP (Pure f) e</td> <td style="padding: 2px 10px 2px 10px;">= BIND ($\lambda x, \text{Pure } (\text{App } f \ x)$) e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>identity law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">AP f (Pure e)</td> <td style="padding: 2px 10px 2px 10px;">= BIND ($\lambda x, \text{Pure } (\text{App } x \ e)$) f</td> <td style="padding: 2px 10px 2px 10px;">-- <i>interchange law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">AP $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">= Ap $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">-- <i>indistinguishable</i></td> </tr> </table> </td> </tr> <tr> <td colspan="3" style="padding: 10px 0 10px 20px;"> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px 2px 10px;">BIND g (Bind $f \ e$)</td> <td style="padding: 2px 10px 2px 10px;">= BIND (Bind $g \circ \text{App } f$) e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>associativity law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">BIND f (Pure e)</td> <td style="padding: 2px 10px 2px 10px;">= App $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">-- <i>left unit law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">BIND Pure e</td> <td style="padding: 2px 10px 2px 10px;">= e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>right unit law</i></td> </tr> </table> </td> </tr> </table>			PURE (Var x)	= Pure (Var x)	-- <i>indistinguishable</i>	PURE (Each e)	= BIND id (PURE e)	-- <i>effect translation</i>	PURE (App $f \ e$)	= AP (PURE f) (PURE e)	-- <i>homomorphism law</i>	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px 2px 10px;">AP (Pure f) e</td> <td style="padding: 2px 10px 2px 10px;">= BIND ($\lambda x, \text{Pure } (\text{App } f \ x)$) e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>identity law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">AP f (Pure e)</td> <td style="padding: 2px 10px 2px 10px;">= BIND ($\lambda x, \text{Pure } (\text{App } x \ e)$) f</td> <td style="padding: 2px 10px 2px 10px;">-- <i>interchange law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">AP $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">= Ap $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">-- <i>indistinguishable</i></td> </tr> </table>			AP (Pure f) e	= BIND ($\lambda x, \text{Pure } (\text{App } f \ x)$) e	-- <i>identity law</i>	AP f (Pure e)	= BIND ($\lambda x, \text{Pure } (\text{App } x \ e)$) f	-- <i>interchange law</i>	AP $f \ e$	= Ap $f \ e$	-- <i>indistinguishable</i>	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px 2px 10px;">BIND g (Bind $f \ e$)</td> <td style="padding: 2px 10px 2px 10px;">= BIND (Bind $g \circ \text{App } f$) e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>associativity law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">BIND f (Pure e)</td> <td style="padding: 2px 10px 2px 10px;">= App $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">-- <i>left unit law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">BIND Pure e</td> <td style="padding: 2px 10px 2px 10px;">= e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>right unit law</i></td> </tr> </table>			BIND g (Bind $f \ e$)	= BIND (Bind $g \circ \text{App } f$) e	-- <i>associativity law</i>	BIND f (Pure e)	= App $f \ e$	-- <i>left unit law</i>	BIND Pure e	= e	-- <i>right unit law</i>
PURE (Var x)	= Pure (Var x)	-- <i>indistinguishable</i>																																	
PURE (Each e)	= BIND id (PURE e)	-- <i>effect translation</i>																																	
PURE (App $f \ e$)	= AP (PURE f) (PURE e)	-- <i>homomorphism law</i>																																	
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px 2px 10px;">AP (Pure f) e</td> <td style="padding: 2px 10px 2px 10px;">= BIND ($\lambda x, \text{Pure } (\text{App } f \ x)$) e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>identity law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">AP f (Pure e)</td> <td style="padding: 2px 10px 2px 10px;">= BIND ($\lambda x, \text{Pure } (\text{App } x \ e)$) f</td> <td style="padding: 2px 10px 2px 10px;">-- <i>interchange law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">AP $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">= Ap $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">-- <i>indistinguishable</i></td> </tr> </table>			AP (Pure f) e	= BIND ($\lambda x, \text{Pure } (\text{App } f \ x)$) e	-- <i>identity law</i>	AP f (Pure e)	= BIND ($\lambda x, \text{Pure } (\text{App } x \ e)$) f	-- <i>interchange law</i>	AP $f \ e$	= Ap $f \ e$	-- <i>indistinguishable</i>																								
AP (Pure f) e	= BIND ($\lambda x, \text{Pure } (\text{App } f \ x)$) e	-- <i>identity law</i>																																	
AP f (Pure e)	= BIND ($\lambda x, \text{Pure } (\text{App } x \ e)$) f	-- <i>interchange law</i>																																	
AP $f \ e$	= Ap $f \ e$	-- <i>indistinguishable</i>																																	
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px 10px 2px 10px;">BIND g (Bind $f \ e$)</td> <td style="padding: 2px 10px 2px 10px;">= BIND (Bind $g \circ \text{App } f$) e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>associativity law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">BIND f (Pure e)</td> <td style="padding: 2px 10px 2px 10px;">= App $f \ e$</td> <td style="padding: 2px 10px 2px 10px;">-- <i>left unit law</i></td> </tr> <tr> <td style="padding: 2px 10px 2px 10px;">BIND Pure e</td> <td style="padding: 2px 10px 2px 10px;">= e</td> <td style="padding: 2px 10px 2px 10px;">-- <i>right unit law</i></td> </tr> </table>			BIND g (Bind $f \ e$)	= BIND (Bind $g \circ \text{App } f$) e	-- <i>associativity law</i>	BIND f (Pure e)	= App $f \ e$	-- <i>left unit law</i>	BIND Pure e	= e	-- <i>right unit law</i>																								
BIND g (Bind $f \ e$)	= BIND (Bind $g \circ \text{App } f$) e	-- <i>associativity law</i>																																	
BIND f (Pure e)	= App $f \ e$	-- <i>left unit law</i>																																	
BIND Pure e	= e	-- <i>right unit law</i>																																	

■ **Figure 1** Optimised Translation.

parallelism. The optimisation rules that apply to BIND (Figure 1) are similar. If either of its arguments is marked as pure, we can avoid performing effects at all. If we have nested binds, we can apply the associativity rule to generate a chain of binds.

Overall, seven of the ten equations above come from our generalized laws; two hold by semantic *indistinguishability*, and one is the basis for our *effect translation*, namely the translation of the imperative \leftarrow to an explicit bind.

In the following section, we extend the language, formalize the language and the translation using a Coq mechanisation, and prove correctness.

3 Mechanisation

We define the source language that features our effect notation and a translation to a target language which is the subset of the source language that does not include the effect notation. We prove that our translation preserves typability, semantics, and parallel execution, which we measure through the program's span and work. We have mechanized our language and proofs in Coq.

3.1 Definitions

We use (parametric) higher-order abstract syntax (PHOAS) [40, 8], which enables us to reuse the binders of the host language as binders of the guest language. PHOAS avoids the need to define first-order syntax, an operational semantics and capture-avoiding substitution, thereby removing intricate lemmas regarding substitution and hundreds of lines of code from the mechanisation, bringing the proof more in line with a more legible pen-and-paper formalisation.

Further, we use intrinsically typed terms [11, 3, 1, 2], and a type-theoretic semantics [17]. Using intrinsically typed terms together with dependent pattern matching allows us to define total evaluation (in contrast to using untyped terms or simple pattern matching where we could just define partial evaluation). The reason is that such an approach only needs to consider well-formed terms that *don't go wrong* [33].

■ **Listing 1** Lawful Monad.

```

Class Monad F := {
  map {A B}: (A → B) → F A → F B;
  pure {A} : A → F A;
  ap {A B}: F (A → B) → F A → F B;
  bind {A B}: (A → F B) → F A → F B;
}.

Class LawfulMonad F := {
  monad :> Monad F;

  idl {A B} (f: A → F B) {x}: bind f (pure x) = f x;
  idr {A B} (f: A → B) {x}: bind (pure ∘ f) x = map f x;
  asc {A B C} (f: A → F B) (g: B → F C) {x}: bind g (bind f x) = bind (bind g ∘ f) x;

  apl {A B} (f: A → B) {x}: ap (pure f) x = map (λ x', f x') x;
  apr {A B} (f: F (A → B)) {x}: ap f (pure x) = map (λ f', f' x) f;
  aplr {A B} (f: A → B) {x}: map f (pure x) = pure (f x);

  map_map {A B C} (g: A → B) (f: B → C) {x}: map f (map g x) = map (λ x, f (g x)) x;
}.

```

Coq

The common strategy behind all these approaches is to carve out a subset of the host language, that is the language we want to define (the guest language), and then reusing all the power of the host language to define the guest language, avoiding having to reimplement tedious implementation details: The guest types simply mirror the host types, the guest terms mirror the host terms, and the evaluation function maps guest terms to host terms.

Lawful monads. For brevity, we do not define Functor, Applicative and Monad separately. We define a class `Monad` and a class `LawfulMonad` (Listing 1). `Monad` contains the functions `map`, `pure`, `ap`, and `bind`. `LawfulMonad` extends `Monad` and further contains `idl`, `idr`, `asc`, `apl`, `apr`, `aplr`, and `map_map`, corresponding to the left and right unit law, and the associativity law of the monad, and the identity and interchange law of the applicative, the free theorem of pure, and the composition law of the functor.

Static semantics. From Coq, we use units ($\text{tt}: \text{Unit}$), products $((a,b): A \times B)$, functions $((\lambda a, b): A \rightarrow B)$. Mirroring the data types of the host language, we define the types for unit (\mathbb{T}), sums $(s \vee t)$, products $(s \wedge t)$, functions $(s \rightsquigarrow t)$ and effects (\mathbb{M}) in the guest language (Listing 2a). We define a data type `ef` to label terms with, as belonging to the source language `src`, the target language `tgt`, or the either language `com` (common) (Listing 2c). Label denotation $\text{EF } m: \text{ef} \rightarrow (\text{Type} \rightarrow \text{Type})$ assigns each functor in the host language. Concretely, the target and common label is assigned the identity effect functor (e.g. no effect), and the source language is assigned the effect `m` given as an argument.

We define a data type `tm` $\Gamma \text{ B } t$ for the syntax of our guest language (Listing 2d). The terms are parametrized by a type denotation Γ , a language label `B` and a type `t`. The common term formers are abstraction `Lam e`, application `App e f` and variables `Var v` to represent functions; unit `Unt e`, tuple `Prd (e, f)` and projections `Fst e` and `Snd e` to represent products. The source language has an additional term former `Each e`, which represents the direct-style effect application \downarrow from above. The target language has additional term formers `Pure e`, `Ap e`, `Map f e`, and `Join e` representing the effect combinators.

25:12 A Direct-Style Effect Notation for Sequential and Parallel Programs

The term former `Lam` binds variables. In PHOAS, guest-level bindings are represented using the host language's bindings. This is why this construct takes as an argument a function which binds a variable, represented as a value of type Γt .

Example. In our encoding, the terms of the guest language can be written similarly to the terms of the host language where each term former of the host language is wrapped by a term former of the guest language.

For example, the identity function $(\lambda x, x)$ can be encoded in the guest language as `Lam` $(\lambda x, \text{Var } x)$. A term $(\lambda x y, \text{add } x y) a b$ – an eta-expanded addition function applied to some arguments – can be expressed as `Lam` $(\lambda x, (\text{Lam } (\lambda y, \text{add } \text{`App` } (\text{Var } y) \text{`App` } (\text{Var } x)))) \text{`App` } (\text{Var } a) \text{`App` } (\text{Var } b)$, writing application $x \text{`App` } y$ infix for convenience. Constructing a term of unit type `tt` is written in the guest language as `Unt tt`. Similarly, projection on a pair (a,b) is written in the guest language as `Fst (Prd (a,b))`.

Dynamic semantics. Next, we define the dynamic semantics corresponding to the static semantics. The static semantics has three parts: the types, the language labels and the terms. Therefore, the dynamic semantics also defines three parts.

The denotation of a type `EVAL m: ty \rightarrow Type` (Listing 2b) maps each guest type to its corresponding host type, and is parametrized by a type constructor `m`, corresponding to the monad we evaluate in.

The denotation of a term with regard to the previously defined type denotation `eval ... : tm (EVAL m) B t \rightarrow EF m B (EVAL m t)` (Listing 2d) interprets the terms in a specific monad depending on which language the terms are labeled from. More concretely it takes a term of type `t` and of label `B` to be evaluated in monad `m`, and returns a value of the denotation of the type `EVAL m t` wrapped in the denotation of the label `EF m B`. The evaluation for terms from the source language implicitly have effects and can therefore only be interpreted in a monadic interpreter. For the common and the target language, we define an evaluation as simply the mapping of guest term formers to their corresponding host expressions, while mapping variables to variables.

The decision of which monad to use is governed by the label denotation `EF m: ef \rightarrow ty \rightarrow ty` (Listing 2c) mapping the target and the common language (whose terms do not have implicitly any effects) to the identity effect, e.g., no effect, while the `src` language is mapped to the effect `M`.

Note the way we defined the common, source and target terms, we can relate common terms into source or target terms, e.g., into any other language label `relabel: T Γ com t \rightarrow T Γ e t`.

Example. Consider the evaluation of the following term, which constructs and then destructs a pair of units, which is equal to unit: `eval com (Fst (Prd (Unt tt, Unt tt))) = tt`.

Translation. The compilation from the target into the source language is performed by the smart constructor `PURE`, i.e., we compile from an effectful language into a pure language that uses monadic effect combinators. We formally define `PURE` (Listing 3) that performs both the action of a normal `pure`, e.g., wraps the argument into an additional effect `tm Γ src t \rightarrow tm Γ tgt (M t)`, and additionally performs a translation from terms from the source language with effect application `Each` to terms of the target language using combinators `Pure`, `Map`, `Ap` and `Bind`. This translation makes use of the smart constructors `AP` and `JOIN`, that perform optimisations.

■ **Listing 2** Syntax and semantics.

(a) Types.	(b) Type denotation.	(c) Labels and denotation.
<p>Inductive</p> $\text{ty} : \text{Type} :=$ <ul style="list-style-type: none"> $\mathbb{T} : \text{ty}$ $\vee : \text{ty} \rightarrow \text{ty} \rightarrow \text{ty}$ $\wedge : \text{ty} \rightarrow \text{ty} \rightarrow \text{ty}$ $\rightsquigarrow : \text{ty} \rightarrow \text{ty} \rightarrow \text{ty}$ $\mathbb{M} : \text{ty} \rightarrow \text{ty}$. <p style="text-align: right;">Coq</p>	<p>Equations</p> $\text{EVAL} (m : \text{Type} \rightarrow \text{Type}) : \text{ty} \rightarrow \text{Type} :=$ <ul style="list-style-type: none"> $m, \mathbb{T} \Rightarrow \text{Unit}$ $m, s \vee t \Rightarrow \text{EVAL } m \ s \ + \ \text{EVAL } m \ t$ $m, s \wedge t \Rightarrow \text{EVAL } m \ s \ \times \ \text{EVAL } m \ t$ $m, s \rightsquigarrow t \Rightarrow \text{EVAL } m \ s \ \rightarrow \ \text{EVAL } m \ t$ $m, \mathbb{M} \ t \Rightarrow m \ (\text{EVAL } m \ t)$. <p style="text-align: right;">Coq</p>	<p>Inductive</p> $\text{ef} := \text{src} \mid \text{tgt} \mid \text{com}.$ <p>Equations</p> $\text{EF} m : \text{ef} \rightarrow \text{Type} \rightarrow \text{Type} :=$ <ul style="list-style-type: none"> $m, \text{src}, t \Rightarrow m \ t$ $m, \text{com}, t \Rightarrow t$ $m, \text{tgt}, t \Rightarrow t$. <p style="text-align: right;">Coq</p>
<p>(d) Term and their denotation.</p> <hr/> <p>Inductive $\text{tm} \{\Gamma : \text{ty} \rightarrow \text{Type}\} : \text{ef} \rightarrow \text{ty} \rightarrow \text{Type} :=$</p> <ul style="list-style-type: none"> $\text{Var} \{B \ t\} : \Gamma \ t \rightarrow \text{tm} \ B \ t$ $\text{Unt} \{B\} : \text{Unit} \rightarrow \text{tm} \ B \ \mathbb{T}$ $\text{Prd} \{B \ s \ t\} : \text{tm} \ B \ s \ \times \ \text{tm} \ B \ t \rightarrow \text{tm} \ B \ (s \ \wedge \ t)$ $\text{Fst} \{B \ s \ t\} : \text{tm} \ B \ (s \ \wedge \ t) \rightarrow \text{tm} \ B \ s$ $\text{Snd} \{B \ s \ t\} : \text{tm} \ B \ (s \ \wedge \ t) \rightarrow \text{tm} \ B \ t$ $\text{App} \{B \ s \ t\} : \text{tm} \ B \ (s \rightsquigarrow t) \rightarrow (\text{tm} \ B \ s \rightarrow \text{tm} \ B \ t)$ $\text{Lam} \{B \ s \ t\} : (\Gamma \ s \rightarrow \text{tm} \ \text{com} \ t) \rightarrow \text{tm} \ B \ (s \rightsquigarrow t)$ <ul style="list-style-type: none"> $\text{Each} \{t\} : \text{tm} \ \text{src} \ (\mathbb{M} \ t) \rightarrow \text{tm} \ \text{src} \ t$ <ul style="list-style-type: none"> $\text{Pure} \{t\} : \text{tm} \ \text{com} \ t \rightarrow \text{tm} \ \text{tgt} \ (\mathbb{M} \ t)$ $\text{Join} \{t\} : \text{tm} \ \text{tgt} \ (\mathbb{M} \ (\mathbb{M} \ t)) \rightarrow \text{tm} \ \text{tgt} \ (\mathbb{M} \ t)$ $\text{Map} \{s \ t\} : \text{tm} \ \text{tgt} \ (s \rightsquigarrow t) \rightarrow (\text{tm} \ \text{tgt} \ (\mathbb{M} \ s) \rightarrow \text{tm} \ \text{tgt} \ (\mathbb{M} \ t))$ $\text{Ap} \{s \ t\} : \text{tm} \ \text{tgt} \ (\mathbb{M} \ (s \rightsquigarrow t)) \rightarrow (\text{tm} \ \text{tgt} \ (\mathbb{M} \ s) \rightarrow \text{tm} \ \text{tgt} \ (\mathbb{M} \ t))$. <p>Equations $\text{eval} \ \{t \ m\} \ \{M : \text{Monad} \ m\} \ B : \text{tm} \ (\text{EVAL} \ m) \ B \ t \rightarrow \text{EF} \ m \ B \ (\text{EVAL} \ m \ t) :=$</p> <ul style="list-style-type: none"> $\text{src}, \text{Var} \ i \Rightarrow M.(\text{pure}) \ i \quad (* \ \text{src} \ *)$ $\text{src}, \text{Lam} \ k \Rightarrow M.(\text{pure}) \ (\text{eval} \ \circ \ k)$ $\text{src}, \text{Unt} \ tt \Rightarrow M.(\text{pure}) \ tt$ $\text{src}, \text{Fst} \ e \Rightarrow M.(\text{map}) \ (\lambda \ e', \ e'.1) \ (\text{eval} \ e)$ $\text{src}, \text{Snd} \ e \Rightarrow M.(\text{map}) \ (\lambda \ e', \ e'.2) \ (\text{eval} \ e)$ $\text{src}, \text{App} \ e \ f \Rightarrow M.(\text{ap}) \ (\text{eval} \ e) \ (\text{eval} \ f)$ $\text{src}, \text{Prd} \ (e, \ f) \Rightarrow M.(\text{ap}) \ (M.(\text{map}) \ (\lambda \ a' \ b', \ (a', \ b')) \ (\text{eval} \ e)) \ (\text{eval} \ f)$ $\text{src}, \text{Each} \ e \Rightarrow M.(\text{bind}) \ \text{id} \ (\text{eval} \ e)$ <ul style="list-style-type: none"> $_ , \text{Var} \ i \Rightarrow i \quad (* \ \text{com} \ \text{or} \ \text{tgt} \ *)$ $_ , \text{Lam} \ k \Rightarrow \text{eval} \ \circ \ k$ $_ , \text{Fst} \ e \Rightarrow (\text{eval} \ e).1$ $_ , \text{Snd} \ e \Rightarrow (\text{eval} \ e).2$ $_ , \text{App} \ e \ f \Rightarrow (\text{eval} \ e) \ (\text{eval} \ f)$ $_ , \text{Prd} \ (e, \ f) \Rightarrow (\text{eval} \ e, \ \text{eval} \ f)$ $_ , \text{Unt} \ tt \Rightarrow tt$ $\text{tgt}, \text{Map} \ f \ e \Rightarrow M.(\text{map}) \ (\text{eval} \ f) \ (\text{eval} \ e) \quad (* \ \text{only} \ \text{tgt} \ *)$ $\text{tgt}, \text{Ap} \ f \ e \Rightarrow M.(\text{ap}) \ (\text{eval} \ f) \ (\text{eval} \ e)$ $\text{tgt}, \text{Pure} \ e \Rightarrow M.(\text{pure}) \ (\text{eval} \ e)$ $\text{tgt}, \text{Join} \ e \Rightarrow M.(\text{bind}) \ \text{id} \ (\text{eval} \ e)$. <p style="text-align: right;">Coq</p>		

■ **Listing 3** Translation.

Notation "f `AP` e" := (AP f e) (at level 20).

Equations PURE {Γ x} (e: tm Γ src x): tm Γ tgt (M x) :=
 | Var i ⇒ Pure (Var i)
 | Unt tt ⇒ Pure (Unt tt)
 | Lam j ⇒ Pure (Lam j)
 | Fst e ⇒ Pure (Λ e', Fst (Var e')) `AP` PURE e
 | Snd e ⇒ Pure (Λ e', Snd (Var e')) `AP` PURE e
 | Prd (e, f) ⇒ Pure (Λ e' f', Prd (Var e', Var f')) `AP` PURE e `AP` PURE f
 | App e f ⇒ PURE e `AP` PURE f
 | Each e ⇒ JOIN (PURE e).

Equations AP {Γ s t} (f: tm Γ tgt (M (s ~> t))) (e: tm Γ tgt (M s)): tm Γ tgt (M t) :=
 | Pure f, Pure e ⇒ Pure (App f e)
 | Pure f, e ⇒ Map (Lam (λ x, App f (Var x))) e
 | f, Pure e ⇒ Map (Lam (λ x, App (Var x) e)) f
 | f, e ⇒ Ap f e.

Equations JOIN {Γ t} (e: tm Γ tgt (M (M t))): tm Γ tgt (M t) :=
 | Pure e ⇒ to e
 | e ⇒ Join e.

Coq

The Var, App and Each cases were discussed in Section 2.4: The direct-style use of effects Each is expanded into effect operation Bind, while variables are wrapped in PURE, and function application is translated to applicative Ap. The lambda and empty terms describe values and are simply wrapped into a pure as well.

In the case of projections and the case of tuples, we follow the general pattern of the homomorphism law, e.g., we map both the function (projection, tuple) into a Pure and we wrap all arguments in a PURE, and we apply them applicatively.

Example. Assume our language contains an effectful operation fetch. Then, translating the term e := Prd (Each (fetch "foo"), Each (fetch "bar")) yields PURE e = Pure (Lam (λ e', (Lam (λ f', Prd (Var e', Var f')))) `AP` (fetch "foo") `AP` (fetch "bar")).

Span and Work. We define span and work (Listing 4), which we use to express the degree of parallelism. Span is the length of the longest chain of unhandled effectful operations, i.e., the longer the path, the more operations need to run sequentially. Hence, a shorter span for the same number of operations means a higher amount of parallelism. Work is the sum of all unhandled effectful operations. Just like evaluation interprets the value of a term, span and work are interpretations to a numeric value of a term.

As our syntax is defined from types, label and terms, we define these new interpretations as a type denotation, an effect denotation and a term denotation as well. The effect denotation for span and work is the identity function, and the type denotation is the constant function mapping all guest types to the type of natural numbers (SPAN, WORK).

More formally, we define the span of an expression to be zero for variables and values, such as empty and lambda, and for pure expressions. The span of Join and direct-style effect application Each is one more (successor S) than the span of their argument. For assertion and projection (access to first and second component), the span is simply the span of its

■ **Listing 4** Span and Work.

(a) Span.

Equations SPAN: $ty \rightarrow Type := | _ \Rightarrow nat.$

Equations

```
span {B x} (e: tm SPAN B x): nat :=
| Var i   => 0 | Lam e   => 0
| Unt tt  => 0 | Pure e  => 0
| Fst e   => span e
| Snd e   => span e
| Prd (e, f) => max (span e) (span f)
| App e f  => max (span e) (span f)
| Ap e f   => max (span e) (span f)
| Map e f  => max (span e) (span f)
| Join e   => S (span e)
| Each e   => S (span e).
```

Coq

(b) Work.

Equations WORK: $ty \rightarrow Type := | _ \Rightarrow nat.$

Equations

```
work {B x} (e: tm WORK B x): nat :=
| Var i   => 0 | Lam e   => 0
| Unt tt  => 0 | Pure e  => 0
| Fst e   => work e
| Snd e   => work e
| Prd (e, f) => work e + work f
| App e f  => work e + work f
| Ap e f   => work e + work f
| Map e f  => work e + work f
| Join e   => S (work e)
| Each e   => S (work e).
```

Coq

argument, while the span of a tuple is the maximum of its left or right branch. The span for function application, applicative application and mapping is the maximum of the span of its arguments as well, plus the span of the execution of the specified function on the argument. However, we defined our static semantics such that direct-style effect application cannot be performed under a lambda, therefore the span of the execution of any function is zero.

Analogously, we define the work of an expression to be zero for variables, values, and pure expressions. Similar to the span, the work of Join and direct-style effect application Each is one more (successor S) than the work of their argument. The work of assertion and projection is the work of its argument. Other than span (which takes the maximum), the work of a tuple is the sum of both arguments. The work for function application, applicative application and mapping is the sum of the work of its arguments, plus the work of the execution of the specified function on the argument, which is zero, because lambdas cannot contain Join or Each.

Example. Assume our language contains an effectful operation `fetch`. We calculate the span and work of a term in the source language $e := \text{Prd}(\text{Each}(\text{fetch } \text{"foo"}), \text{Each}(\text{fetch } \text{"bar"}))$ as follows: $\text{span } e = 1$ and $\text{work } e = 2$. This expresses the fact that the two effects can be performed in parallel. The corresponding target language term is $e' := \text{Pure}(\text{Lam}(\lambda e', (\text{Lam}(\lambda f', \text{Prd}(\text{Var } e', \text{Var } f'))))) \text{ `AP` } (\text{fetch } \text{"foo"}) \text{ `AP` } (\text{fetch } \text{"bar"})$. We get the same results for this term: $\text{span } e' = 1$ and $\text{work } e' = 2$.

3.2 Proof

Our translation should only change the encoding from direct-style to effect combinators, while the semantics, typability and parallelism of the term should be preserved. We prove that our translation preserves typability, semantics, span and work. Intuitively, the theorems hold, because our translation performed by PURE, AP, and JOIN are the functor, monad and applicative laws.

► **Theorem 3.1** (PURE preserves types). The translation function takes a well-typed term and produces a well-typed term, i.e., $\text{PURE}: \forall t, \text{tm } \Gamma \text{ src } t \rightarrow \text{tm } \Gamma \text{ tgt } (\mathbb{M} t)$

Proof. Using intrinsically-typed representation of terms, the well-typedness of the translated term is guaranteed by the fact that the definition of the translation function PURE is itself well-typed in Coq. ◀

25:16 A Direct-Style Effect Notation for Sequential and Parallel Programs

We now consider the preservation of semantics. First, we show that the semantics of the smart constructors is equal to that of the normal constructors, so that they merely represent optimizations of those.

► **Lemma 3.2** (AP respects semantics). $\forall f\ e, \text{ eval_tgt } (\text{AP } f\ e) = \text{eval_tgt } (\text{Ap } f\ e)$

► **Lemma 3.3** (JOIN respects semantics). $\forall f\ e, \text{ eval_tgt } (\text{JOIN } e) = \text{eval_tgt } (\text{Join } e)$

Proof. By case distinction on the term structure of the arguments, using the functor, monad and applicative laws. ◀

Next, we see that embedding the pure com sublanguage in the target language preserves the semantics:

► **Lemma 3.4** (relabel preserves semantics). $\forall e, \text{ eval_tgt } (\text{relabel } e) = \text{eval_com } e$

Proof. By induction on the structure of e . ◀

From this, we can deduce that the PURE transformation preserves the semantics of the source program.

► **Theorem 3.5** (PURE preserves semantics). *For all lawful monads M to be evaluated in,*
 $\forall e, \text{ eval_tgt } (\text{PURE } e) = \text{eval_src } e$

Proof. By induction on the structure of e , using Lemmas 3.2–3.4. ◀

We now want to show that PURE preserves the work and span of the program. This is similar to semantics preservation, except that the functions we consider map to a monoid (the natural numbers with addition and maximum, respectively) rather than a monad.

We show that AP and JOIN do not increase the span and work of a term, compared to the normal constructors.

► **Lemma 3.6** (AP respects span and work).

$\forall f\ e, \text{ span_tgt } (\text{AP } f\ e) \leq \text{span_tgt } (\text{Ap } f\ e) \quad \text{and} \quad \text{work_tgt } (\text{AP } f\ e) \leq \text{work_tgt } (\text{Ap } f\ e)$

► **Lemma 3.7** (JOIN respects span and work).

$\forall e, \text{ span_tgt } (\text{JOIN } e) \leq \text{span_tgt } (\text{Join } e) \quad \text{and} \quad \text{work_tgt } (\text{JOIN } e) \leq \text{work_tgt } (\text{Join } e)$

Proof. By case distinction on the term structure of the arguments, using the monoid laws. ◀

The pure terms in the com sublanguage are effect-free; therefore, their span and work is equal to 0.

► **Lemma 3.8** (com is effect-free). $\forall e, \text{ span_com } e = 0 \quad \text{and} \quad \text{work_com } e = 0$

Proof. By induction on the term structure of e . ◀

Embedding pure terms into the target language produces a term that does not perform any effects, either.

► **Lemma 3.9** (relabel terms remain effect-free).

$\forall e, \text{ span_tgt } (\text{relabel } e) = 0 \quad \text{and} \quad \text{work_tgt } (\text{relabel } e) = 0$

Proof. By induction on the term structure of e . ◀

We can then show that the translation PURE does not increase the span or work of the source program, thereby demonstrating that it is parallelism-preserving.

► **Theorem 3.10** (PURE preserves span and work).

∀ e, $\text{span } \text{tgt}(\text{PURE } e) \leq \text{span } \text{src } e$ and $\text{work } \text{tgt}(\text{PURE } e) \leq \text{work } \text{src } e$

Proof. By induction on the term structure of e , using the monoid laws of addition and maximum as well as Lemmas 3.6–3.9. ◀

4 Implementation

In this section, we describe the differences and similarities between the mechanisation in Coq and the implementation in Scala built on a macro-based AST transformation.

Structural Recursion. We keep our implementation in a general-purpose language as close to the formal model of our core calculus as possible. To this end, our implementation follows the formal translation as a structurally recursive function over the terms where possible. We use Scala macros to get access to code as AST data type, similar to the `tm` data type in the formalization.

Type-preserving Compilation. In Scala, we process the untyped AST for fine-grained detailed manipulations. Knowing that the translation is typability-preserving by our Coq proof, increases confidence in the implementation.

Exhaustiveness Checks. A difference between the Coq and the Scala implementation is that `Bind`, `Ap` and `Pure` are not syntax forms in Scala but represented by variable and function application in the embedding. Still, we can treat them as syntax forms to construct and destruct by defining custom patterns for pattern matching. Further, Scala macros do not define the Scala syntax as an algebraic data type (to hide compiler internals), and therefore do not offer exhaustiveness checks. Yet, by the fact that the Coq implementation is total, the Scala implementation can be expected to be as well.

Custom Effects. In the formalization, we have only a single effect, while, in the implementation, we allow every use of the notation to be instantiated with a different effect, based on the type of the expression. Our macro inspects the expression’s type and, based on this type, picks the corresponding generated combinators `bind/ap/pure` of the respective effect.

Arity. In Scala, functions may take multiple arguments. Generally, we can model functions taking multiple arguments as functions taking a single argument of a tuple with multiple fields with appropriate currying and uncurrying. Functions with multiple arguments make the compiler no longer structurally recursive over terms because, besides terms, the compiler additionally needs to mutually recurse over the list of arguments, which would complicate the proof, but is necessary for our implementation.

5 Related work

Do-Notation. Do-notations have been popular for studying a variety of styles for writing effectful code: Wadler extends list-comprehension syntax [55] to monadic comprehensions, from which modern do-notation sprung, and McBride introduced applicatives and idiom brackets as a notation for applicatives [29]. To the best of our knowledge, the only support for mixed sequential and parallel programming was introduced as a Haskell extension [27, 28] to optimise do-notation into mixed monadic/applicative operations (`ApplicativeDo`). In contrast, our notation preserves the parallelism inherent in the structure of the program, thereby allowing sequentiality where necessary and giving parallelism where possible.

Implementations. Besides theory, implementations for effectful guest language notations are a popular endeavor, for example: In Scala, we can find projects to supports effectful programs through compiler plugins such as coroutines [47], Scala async [44], Monadless [7], Effectfull [10], Scala Workflow [49], Scala ContextWorkflow [22], Scala Computation Expressions [46], Dsl.scala [57], Dotty CPS [50]. In other languages we have: F# computation expressions [39], In particular proof-assistants and dependently typed languages have an interest for good support of notations for guest languages, which we can see in Idris’ [6, 21] Lean’s [52, 25], and Kind’s [24] notation. None of them support parallelism.

Further, the following approaches are similar to `ApplicativeDo`: OCaml’s monadic and applicative `let` [37], Scala `avocADO` [45], and Scala `parallel-for` [48]. But these do not support direct-style effect usage, and do not preserve parallelism.

CPS Translations. In general, effects are implemented by translating to other already known effects. In particular, all effects can be represented by the continuation effect [14], and thus, by translating to continuation passing style (CPS) [42, 15]. However, naive CPS translations introduce so called administrative redexes, e.g., expressions containing subexpressions which do not need to be evaluated at run-time, but can already be optimised by a partial evaluation pass at compile-time. Eventually, Danvy and Nielsen [12] optimised the CPS-translation into a first-order, one-pass, compositional translation.

Their trick for achieving an optimal result in one pass is to build optimisations into the definitions of their translation functions. We use a similar approach in our translation through the definition of smart constructors which simplify terms using monad and applicative laws when called.

Host supporting effects. Because effects can be implemented by translation to equally or more powerful effects, besides giving a denotational semantics modelling a compiler, there is another approach – that we did not follow – by forwarding effects to the host language as well. Then, compile-time translations like ours can be avoided and effects can be implemented in languages as a library, given the host languages has sufficient powerful effects. Filinski [14] studied the implementation of effects in languages with delimited continuations (e.g. `shift` and `reset`). In such an impure language it is possible to implement so called monadic reflection – a function taking an effectful function and returning a “pure” function. This is of course only possible by exploiting the impurity of the host language to implement the effect using delimited continuation. Later, Forster [16] studied the translations between monadic reflection, effect handlers and delimited control. The approach to extend the underlying virtual machine by support for delimited continuations, which are sufficiently efficient for then implementing effects as normal libraries is followed by: the JVM proposal for delimited continuations [41], the Haskell proposal for continuation marks [18] and `multicore-ocaml` [35].

We are looking for a more general solution for compiling a language, that works independent of whether the runtime already supports delimited continuations or not.

Formalisation Techniques. To focus on the interesting parts of our formalisation, we used modern techniques to define features of the guest language in terms of features of the host language: In particular, we use parametric higher order abstract syntax (PHOAS) [40, 19, 8] to inherit binders and capture-avoiding substitution from the host language, and intrinsically-typed syntax [11, 3, 1, 2] to inherit type checking. The choice of PHOAS implies a limitation of our work, namely that we can formally only prove theorems about closed terms. Yet, this is a common restriction and lifting it is subject to future work.

6 Conclusion

Existing notations for composing effectful computations fall short on providing both sequential and parallel composition of effects at the same time. In this paper, we proposed a notation for mixed sequential/parallel code. Our notation allows direct-style effects, a feature that enables the sequentiality or parallelism of the effects to be determined by the structure of the code. We proved that our compilation preserves the parallelism of the source program and mechanized the proof in Coq.

An interesting next step for this line of research on direct-style notations for effects is to investigate how to cover more programming language features such as loops and branches, to integrate effects more seamlessly into the language. Besides monad and applicative functors, other effect functors, such as selectives [34, 56], comonads [38], and the theory behind effectful recursion [13] and generalizations such as arrows [20, 26] are promising possibilities.

References

- 1 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Relative monads formalised. *Journal of Formalized Reasoning*, 7(1):1–43, 2014. doi:10.6092/issn.1972-5787/4389.
- 2 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015. doi:10.2168/LMCS-11(1:3)2015.
- 3 Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. Autom. Reason.*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0.
- 4 Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Lost in translation: formalizing proposed extensions to C#. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 479–498. ACM, 2007. doi:10.1145/1297027.1297063.
- 5 Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause 'n' play: Formalizing asynchronous C#. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 233–257. Springer, 2012. doi:10.1007/978-3-642-31057-7_12.
- 6 Edwin C. Brady. Resource-dependent algebraic effects. In Jurriaan Hage and Jay McCarthy, editors, *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers*, volume 8843 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2014. doi:10.1007/978-3-319-14675-1_2.
- 7 Flavio W. Brasil and Sameer Brenn. Monadless - syntactic sugar for monad composition in Scala. <https://github.com/monadless/monadless>.
- 8 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi:10.1145/1411204.1411226.
- 9 Coq 8.16 reference manual. <https://coq.github.io/doc/v8.16/refman/>.
- 10 Tom Crockett. Effectful: A syntax for typeful effectful computations in Scala. <https://github.com/pelotom/effectful#effects-within-conditionals>. Accessed 20-11-2020.
- 11 Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006. doi:10.1007/978-3-540-74464-1_7.
- 12 Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. *Theor. Comput. Sci.*, 308(1-3):239–257, 2003. doi:10.1016/S0304-3975(02)00733-8.

- 13 Levent Erkök and John Launchbury. A recursive do for Haskell. In Manuel M. T. Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, Pittsburgh, Pennsylvania, USA, October 3, 2002*, pages 29–37. ACM, 2002. doi:10.1145/581690.581693.
- 14 Andrzej Filinski. Representing monads. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 446–457. ACM Press, 1994. doi:10.1145/174675.178047.
- 15 Michael J. Fischer. Lambda calculus schemata. In *Proceedings of ACM Conference on Proving Assertions About Programs, Las Cruces, New Mexico, USA, January 6-7, 1972*, pages 104–109. ACM, 1972. doi:10.1145/800235.807077.
- 16 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Journal of Functional Programming*, 29:e15, 2019. doi:10.1017/S0956796819000121.
- 17 Robert Harper and Christopher A. Stone. A type-theoretic interpretation of Standard ML. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 341–388. The MIT Press, 2000.
- 18 Haskell Proposals. Delimited continuation primops. <https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0313-delimited-continuation-primops.rst>.
- 19 Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 204–213. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782616.
- 20 John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000. doi:10.1016/S0167-6423(99)00023-4.
- 21 The Idris Tutorial. Interfaces. Monads and do-notation. !-notation. <http://docs.idris-lang.org/en/latest/tutorial/interfaces.html#notation>. Accessed 14-11-2020.
- 22 Hiroaki Inoue, Tomoyuki Aotani, and Atsushi Igarashi. Contextworkflow: A monadic DSL for compensable and interruptible executions. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPICs*, pages 2:1–2:33. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.ECOOP.2018.2.
- 23 Jerzy Karczmarczuk. Functional differentiation of computer programs. In Matthias Felleisen, Paul Hudak, and Christian Queindec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, pages 195–203. ACM, 1998. doi:10.1145/289423.289442.
- 24 Github. Kind2. A next-gen functional language. <https://github.com/Kindelia/Kind2>. Accessed 29-11-2022.
- 25 Lean Manual. The do notation. Nested actions. <https://leanprover.github.io/lean4/doc/do.html#nested-actions>. Accessed 29-11-2022.
- 26 Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 229(5):97–117, 2011. doi:10.1016/j.entcs.2011.02.018.
- 27 Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. There is no fork: an abstraction for efficient, concurrent, and concise data access. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 325–337. ACM, 2014. doi:10.1145/2628136.2628144.
- 28 Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. Desugaring Haskell's do-notation into applicative operations. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 92–104. ACM, 2016. doi:10.1145/2976002.2976007.
- 29 Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. doi:10.1017/S0956796807006326.

- 30 Erik Meijer. Confessions of a used programming language salesman. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 677–694. ACM, 2007. doi:10.1145/1297027.1297078.
- 31 Erik Meijer. Your mouse is a database. *Communications of the ACM*, 55(5):66–73, 2012. doi:10.1145/2160718.2160735.
- 32 Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706. ACM, 2006. doi:10.1145/1142473.1142552.
- 33 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.
- 34 Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jérémie Dimino. Selective applicative functors. *Proceedings of the ACM on Programming Languages*, 3(ICFP):90:1–90:29, 2019. doi:10.1145/3341694.
- 35 Multicore OCaml. <https://github.com/ocaml-multicore/ocaml-multicore>.
- 36 Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 221–232. ACM, 2005. doi:10.1145/1040305.1040324.
- 37 OCaml: Add "monadic" let operators. <https://github.com/ocaml/ocaml/pull/1947>, 2018.
- 38 Dominic A. Orchard and Alan Mycroft. A notation for comonads. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012. doi:10.1007/978-3-642-41582-1_1.
- 39 Tomas Petricek and Don Syme. The F# computation expression zoo. In *PADL*, volume 8324 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014.
- 40 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 199–208. ACM, 1988. doi:10.1145/53990.54010.
- 41 Project Loom: Fibers and Continuations for the Java Virtual Machine. <https://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html>.
- 42 John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972. doi:10.1145/800194.805852.
- 43 David Richter, David Kretzler, Pascal Weisenburger, Guido Salvaneschi, Sebastian Faust, and Mira Mezini. Prisma: A tierless language for enforcing contract-client protocols in decentralized applications (extended version). *CoRR*, abs/2205.07780, 2022. doi:10.48550/arXiv.2205.07780.
- 44 Scala async rfc. <http://docs.scala-lang.org/sips/pending/async.html>.
- 45 avocADO. Safe compile-time parallelization of for-comprehensions for Scala 3. <https://github.com/kitlangton/parallel-for>.
- 46 Scala Computation Expressions. An implementation of Computation Expressions in Scala. <https://github.com/jedesah/computation-expressions>.
- 47 Coroutines is a library-level extension for the Scala programming language that introduces first-class coroutines. <https://github.com/storm-enroute/coroutines>, 2015.

25:22 A Direct-Style Effect Notation for Sequential and Parallel Programs

- 48 parallel-for. Automatically parallelize your for-comprehensions at compile time. <https://github.com/kitlangton/parallel-for>.
- 49 Scala workflow. Boilerplate-free syntax for computations with effects. <https://github.com/aztek/scala-workflow>.
- 50 Ruslan Shevchenko. dotty-cps-async - experimental CPS transformer for dotty. <https://github.com/rssh/dotty-cps-async>.
- 51 Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2011. doi:10.1007/978-3-642-18378-2_15.
- 52 Sebastian Ullrich and Leonardo de Moura. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proceedings of the ACM on Programming Languages*, 6(ICFP):512–539, 2022. doi:10.1145/3547640.
- 53 Janis Voigtländer. Free theorems simply, via dinaturality. In Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel, editors, *Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9-12, 2019, Revised Selected Papers*, volume 12057 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2019. doi:10.1007/978-3-030-46714-2_16.
- 54 Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi:10.1145/99370.99404.
- 55 Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. doi:10.1017/S0960129500001560.
- 56 Jamie Willis, Nicolas Wu, and Matthew Pickering. Staged selective parser combinators. *Proceedings of the ACM on Programming Languages*, 4(ICFP):120:1–120:30, 2020. doi:10.1145/3409002.
- 57 Bo Yang. Dsl.scala – A framework to create embedded domain-specific languages in Scala. <https://github.com/ThoughtWorksInc/Dsl.scala>.

Sinatra: Stateful Instantaneous Updates for Commercial Browsers Through Multi-Version eXecution

Ugnius Rumsevicius ✉

University of Illinois at Chicago, IL, USA

Siddhanth Venkateshwaran ✉

University of Illinois at Chicago, IL, USA

Ellen Kidane ✉

University of Illinois at Chicago, IL, USA

Luís Pina ✉ 

University of Illinois at Chicago, IL, USA

Abstract

Browsers are the main way in which most users experience the internet, which makes them a prime target for malicious entities. The best defense for the common user is to keep their browser always up-to-date, installing updates as soon as they are available. Unfortunately, updating a browser is disruptive as it results in loss of user state. Even though modern browsers reopen all pages (tabs) after an update to minimize inconvenience, this approach still loses all local user state in each page (e.g., contents of unsubmitted forms, including associated JavaScript validation state) and assumes that pages can be refreshed and result in the same contents. We believe this is an important barrier that keeps users from updating their browsers as frequently as possible.

In this paper, we present the design, implementation, and evaluation of SINATRA, which supports instantaneous browser updates that do not result in any data loss through a novel Multi-Version eXecution (MVX) approach for JavaScript programs, combined with a sophisticated proxy. SINATRA works in pure JavaScript, does not require any browser support, thus works on closed-source browsers, and requires trivial changes to each target page, that can be automated. First, SINATRA captures all the non-determinism available to a JavaScript program (e.g., event handlers executed, expired timers, invocations of `Math.random`). Our evaluation shows that SINATRA requires 6MB to store such events, and the memory grows at a modest rate of 253KB/s as the user keeps interacting with each page. When an update becomes available, SINATRA transfer the state by re-executing the same set of non-deterministic events on the new browser. During this time, which can be as long as 1.5 seconds, SINATRA uses MVX to allow the user to keep interacting with the old browser. Finally, SINATRA changes the roles in less than 10ms, and the user starts interacting with the new browser, effectively performing a browser update with zero downtime and no loss of state.

2012 ACM Subject Classification Computer systems organization → Availability; Software and its engineering → Maintaining software

Keywords and phrases Internet browsers, dynamic software updating, multi-version execution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.26

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.3>

Funding This work was funded in part by NSF CCF-2227183.

1 Introduction

Browsers are the main way in which most users experience the internet. Browsers are responsible for the safety of user sensitive data, in the form of cookies, saved passwords, and credit card information, and other personal information used to auto-complete forms.



© Ugnius Rumsevicius, Siddhanth Venkateshwaran, Ellen Kidane, and Luís Pina;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 26; pp. 26:1–26:29



Leibniz International Proceedings in Informatics
LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Browsers are also responsible for ensuring the integrity of the websites that the user visits, checking certificates and negotiating encrypted HTTPS channels. Given all this, browsers are prime targets for malicious entities. For the common user, the best way to protect their browsers (and the personal data they keep) is to keep the browsers as up-to-date as possible.

Unfortunately, users are slow to update their browser to a new version. In terms of percentage of users, data for Google Chrome [53] and Mozilla Firefox [41] show that a new browser version takes about 2 weeks to overtake the previous version, and about 4 weeks to reach its peak. Given the fast pace of browser releases (6 weeks for Google Chrome and 4 weeks for Mozilla Firefox), the amount of users running outdated versions is significant at any given time.

Browser developers are aware of the problems caused by running outdated versions, and provide features to entice users to update, from reminding the user that a new update is available to minimizing the inconvenience by reopening all pages (tabs) after the update. Even though popular, the latter feature has three main flaws. **First**, it disrupts the user interaction by closing the browser, downloading the new version, and then opening it. **Second**, it assumes that pages can simply be refreshed after the update. Such an assumption fails if a login session expires, which causes the page to refresh to the login portal; or if the contents of the page change with each refresh, as is the case with modern social media. **Third**, refreshing a page loses all user state accumulated on that page since it was loaded. Such state includes, among others, data in HTML forms and JavaScript state.

The result is simple: **Browser updates are disruptive for the average user**. Dynamic Software Updating (DSU) techniques can be used for eliminating such disruption, updating a program in-process. Unfortunately, state-of-the-art DSU tools cannot handle programs as complex as modern commercial internet browsers (Section 2.1). Also, simply dumping the old browser memory state to disk and reloading it in the new browser does not work, as the new browser may change the internal state representation.

In this paper, we present the design and implementation of **Sinatra— Stateful Instantaneous browser updates** – a novel MVX technique implemented in pure JavaScript. SINATRA requires little changes to the target JavaScript application (Section 3), which can be performed automatically for all the pages accessed through an HTTP proxy (Section 3.1).

To perform an update (Section 3.2), SINATRA captures all sources of non-determinism accessed by the browser. Then, when an update becomes available, SINATRA launches the updated browser as a separate process, and feeds it the same non-determinism, thus synchronizing the JavaScript state between both browsers. During this time, SINATRA allows the user to keep interacting with the old browser by performing MVX until the updated browser's state is up-to-date. Once the update was successful, SINATRA terminates the old browser and the user can start interacting with the new browser.

Note that simply transferring the JavaScript between browsers is not sufficient for two reasons. First, the user cannot interact with either browser while transferring the state. Second, failed updates may still result in loss of user data. Multi-Version eXecution (MVX) solves both problems by allowing the user to interact with the old browser while the new browser is receiving the state, and by allowing SINATRA to cancel a failed update simply by closing the new browser. Unfortunately, state-of-the-art MVX tools cannot handle modern commercial internet browsers (Section 2.2), and performing MVX at the JavaScript is not as straightforward due to the event-driven programming paradigm (Section 2.3).

SINATRA captures all sources of non-determinism available to a JavaScript program, including execution of event handlers (Sections 3.3), and non-deterministic functions such as `Math.random` (Section 3.4). We implemented SINATRA in pure JavaScript using an extra

coordinator process to enable communication between browsers (Section 4.1) that serializes JavaScript non-determinism as JSON (Section 4.2). Our implementation also handles all other sources of state in a JavaScript application (Sections 4.2 and 4.3).

This paper also presents an extensive evaluation of SINATRA using 4 JavaScript applications and realistic workloads (Section 6.1) and 2 real-world modern websites (Section 6.8). Our results show that SINATRA runs with very little performance overhead, adding at most 1.896ms to the execution of event handlers (Section 6.2), which is not noticeable by the user. For realistic user interactions, SINATRA requires less than 6MB of memory to store the events until a future update happens (Section 6.3). Furthermore, the amount of memory grows constantly with the length of active user interactions, with a maximum rate of 253KB/s (Section 6.4), which shows that SINATRA scales well with typical user interactions with modern websites. For websites that make use of frequent XML HTTP Requests (XHR) in the background, SINATRA requires a modest 36MB of storage for a 14h run (Section 6.6). Furthermore, SINATRA supports realistic workloads on modern websites as complex as Twitter, with complex JavaScript that requires over 4500 events to load (Section 6.8).

When performing an update, SINATRA requires at most 1.5 seconds to transfer the state between browsers (Section 6.5.1). We note that the user can continue to interact with the browser during this time. To switch to the updated browser, SINATRA imposes a pause in user interaction of less than 10ms (Section 6.5.2), which is perceived as instantaneous. At its core, SINATRA is an MVX system that delivers events from one browser to another in 19ms or less (Section 6.7).

In short, this paper has the following contributions:

1. The design, and implementation of SINATRA, a system for performing MVX on JavaScript applications.
2. A technique to use SINATRA to perform instantaneous updates to modern commercial closed-source internet browsers, without any loss of state.
3. An extensive evaluation of SINATRA using 4 realistic JavaScript stateful applications and 2 popular websites (Google and Twitter); including widely used JavaScript frameworks Angular [21], JsAction [23], and React [38].

SINATRA's source [1] and research artifact [50] are freely available.

2 Background

Performing *Dynamic Software Updating (DSU)* on a running browser presents many unique challenges. First, state-of-the-art DSU tools require source code changes and do not support programs as complicated as modern internet browsers (Section 2.1). SINATRA circumvents that problem by using *Multi-Version eXecution (MVX)* to perform DSU [44]. Unfortunately, state-of-the-art MVX tools also do not support programs as complicated as modern internet browsers (Section 2.2). SINATRA moves the level of MVX from low-level system calls to high-level JavaScript events. However, performing MVX in the traditional sense is not possible in JavaScript, due to its event-driven paradigm (Section 2.3).

2.1 Dynamic Software Updating (DSU)

Dynamic Software Updating (DSU) allows to install an update on a running program without terminating it, and without losing any program state (e.g., data in memory, open connections, open files). DSU has three fundamental problems to solve: (1) when to stop the running program, (2) how to transform the program state to a representation that is *compatible* with

the new version but *equivalent* to the old state, and (3) how to restart the program in the new version. Solving these problems requires modifying the source code of the updatable program [47, 24, 44] adding *safe-update points* to solve problem 1, *state transformation functions* to solve problem 2, and *control-flow migration* to solve problem 3. This is not possible for popular modern internet browsers (e.g., Google Chrome, Microsoft Edge, Apple Safari), as they are closed-source.

Modern DSU approaches focus on *in-process updates* – the new version of the program replaces the old version in the same process – which trivially keep outside resources available between updates (e.g., open network connections and files), but limits existing DSU tools to programs that execute in a single process. This is not the case of modern internet browsers (e.g., Google Chrome uses one process per open tab to improve performance and provide strong isolation between open pages). Finally, modern internet browsers are examples *self-modifying code* given their Just-In-Time (JIT) JavaScript compiler, which is a well known limitation of state-of-the-art DSU tools [24, 44]. Therefore, existing DSU tools cannot update modern browsers.

2.2 Multi-Version eXecution (MVX)

The main goal of MVX is to ensure that many program *versions* execute over the same inputs and generate the same outputs. MVX can be used to perform DSU by launching the updated program as a separate process, transferring the state between processes (e.g., by forking the original process), and resuming execution on the updated process after terminating the outdated process [44].

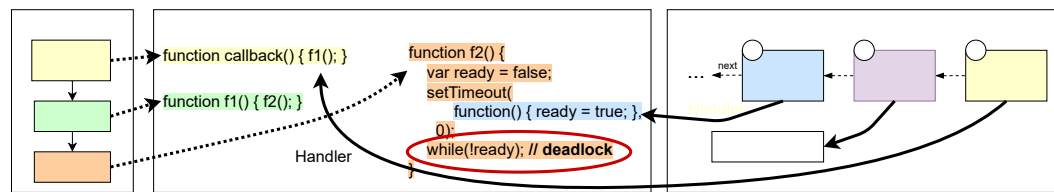
Unlike DSU, state-of-the-art MVX techniques do not require access to the source code of the target program. Instead, MVX interposes *system-calls* through ptrace [35] or binary-code instrumentation [27, 43]. This way, MVX tools can ensure that all processes read the same data, by capturing relevant system-calls (e.g., read) and ensuring that they return the same sequence of bytes.

Unfortunately, existing MVX tools cannot be applied to modern internet browsers. Doing so results in immediate termination due to *benign divergences* – equivalent behavior expressed by different sequences of system calls. For instance, consider how a JIT compiler decides which code to compile/optimize using performance counters based on CPU time. Interacting with such counters does not result in system-calls, and causes JIT compilers to optimize different code, which then results in different system calls. It is possible to tolerate such benign divergences [46], but doing so requires developer support and significant engineering effort, which is not practical.

MVX also suffers from some of the same issues as DSU: no support for multi-process applications, and no support for self-modifying code.

2.3 JavaScript Messages, Event-Loop, and Non-Determinism

JavaScript [16] is an event-driven programming language animated by an *event loop*, as depicted in Figure 1, which processes *messages* from an *event queue*. The event loop takes one message from the event queue and executes its *handler* to completion. If the queue is empty, the event-loop simply waits for the next event. A handler is a JavaScript closure associated with each message. Given that the event loop is *single-threaded*, there is a single *call stack* and one *program counter* (not depicted) to keep the state of processing the current event.



■ **Figure 1** JavaScript's event loop processing 3 messages. Processing Message 1 causes Message 3 to be added to the queue. Message 2 was added when a button was clicked while processing Message 1. Due to JavaScript's event-driven model, this example will never process Message 3, causing a deadlock.

On a browser, events can come from two sources: (1) user interaction with DOM elements (e.g., `onclick` on a `button` element), and (2) browser-generated events, such as expiring timers (e.g., `setTimeout` or `setInterval`) or receiving replies to pending XML HTTP requests. Each event generates a message that keeps track of the event details: the event handler (a closure to be executed for that event), the event target (e.g., the DOM element that generated the message), and other properties. Events handlers in JavaScript are not executed immediately when the event is triggered. Instead, each event is added to the end of the event queue as it is triggered. For instance, in Figure 1, a button was clicked while running function `callback`, which results in adding Message 2 to the event queue.

Events are processed by a single-threaded *event-loop* that runs each event handler to completion before processing any other event, which has two important consequences. **First**, the order and types of events processed are a major part of the non-determinism used to execute a JavaScript program. Apart from asynchronous non-determinism, described below, rerunning the same events in the same order results in the same execution of the same JavaScript program [39, 10, 5, 25, 57]. **Second**, it is not possible for an event handler to issue an event and wait for its completion. This causes the code in Figure 1 to *deadlock* when waiting for the flag `ready` to become `true` [6] because the handler that sets the flag never executes. The handler is associated with a timeout (of zero), which adds a message to the end of the queue. The event loop never finishes executing the current handler, so it never processes any more messages on the queue.

Besides the *synchronous non-determinism* created by events, described above, a JavaScript program can also call functions that are non-deterministic, which we call *asynchronous non-determinism*. The main non-deterministic functions are `Math.random`, which generates random numbers between 0 and 1; and methods of the `Date` object (e.g., `Date.getTime`), which access the current time and date. Notably, it is not possible to seed the pseudo random number generator behind `Math.random`.

Given JavaScript's limitations, **it is not possible to perform traditional MVX on the asynchronous non-determinism**. For instance, when generating a random number, typical MVX approaches ensure each version waits for a message with the same random number (perhaps from a central coordinator process). In JavaScript's case, this would create the same deadlock as shown above. Section 3.4 describes how SINATRA overcomes this limitation.

3 Sinatra Design

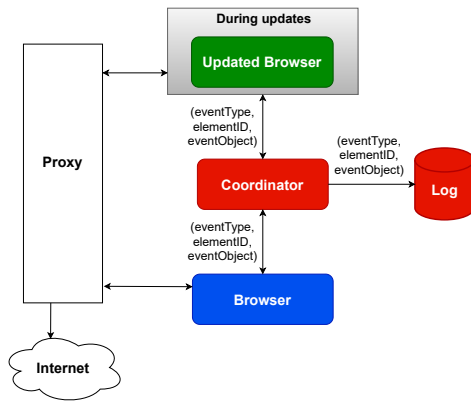
SINATRA supports updating internet browsers through a combination of MVX and DSU [44], both at the JavaScript level. SINATRA requires trivial modifications to web pages, which are shown in Figure 2 – Lines 3–4 need to be added. The required changes ensure that

```

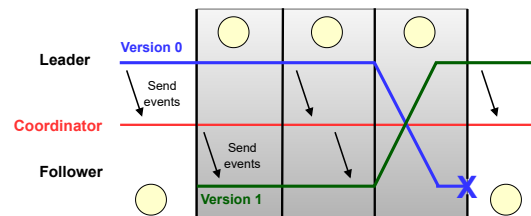
01: <html>
02:   <head>
03: +   <script src="mvx/sinatra.js" type="text/javascript"></script>
04: +   <script src="mvx/sinatra_init.js" type="text/javascript" defer></script>
05:   </head>
06:   <body>
07:     <button id="button" onclick="console.log('DOM0 inline')"></button>
08:     <input id="input_textbox" type="text" />
09:
10:     <script>
11:       let button = document.getElementById("button");
12:       button.onmouseover = function(event) { console.log("DOM0"); };
13:
14:       let textbox = document.getElementById('input_textbox');
15:       let closure = function(ev) { console.log('DOM2'); };
16:       textbox.addEventListener('change', closure);
17:     </script>
18:   </body>
19: </html>

```

■ **Figure 2** Sample HTML code. SINATRA requires adding Lines 3–6.



■ **Figure 3** SINATRA architecture.



■ **Figure 4** SINATRA update phases. Most of the time is spent in Phase 1. Phase 2 transfers state to an updated browser. Phase 2* is optional, and allows to validate if the update was successful. Phase 3 exposes the updated browser to the user.

SINATRA intercepts event handlers immediately (Line 3) and executes its initialization after the page is loaded but before any other JavaScript code executes (due to the `defer` attribute in Line 4). We note that these are simple modifications that can be performed automatically by a sophisticated proxy [12], as we describe in Section 3.1.

After applying the required changes, SINATRA leverages the first-class nature of functions in JavaScript, and replaces a number of important functions to intercept all sources of non-determinism: `[HTMLElement, HTMLDocument].prototype.addEventListener`, `Math.random`, `setTimeout`, `setInterval`, and others; which we describe in Sections 3.3 through 3.5.

3.1 Sinatra Architecture

SINATRA uses three components at all times, shown in Figure 3: (1) the browser, (2) the coordinator, and (3) a proxy. When a browser update is available, SINATRA requires the new version of the browser to be installed at the same time as the old (current) version. The updated browser becomes, temporarily, SINATRA’s fourth component. We note that modern browsers can have multiple versions installed side-by-side by performing manual installation into different folders.

Users interact with the browser, which captures JavaScript events and sends them to the coordinator. The coordinator either saves those events in a log, when there is no update taking place, or sends them to the updated browser, thus performing MVX. Users only start interacting with the updated browser after the update is complete, as we describe in Section 3.2.

The proxy serves three main purposes. **First**, the proxy ensures that the updated browser accesses exactly the same resources as the original browser did. Accessing different resources means that each browser processes different JavaScript events and leads to benign divergences, as described in Section 2.2. **Second**, the proxy ensures that outgoing connections remain open while SINATRA changes the roles of the browsers. This is important to ensure that responses to XML HTTP requests are not lost, which can result in errors in the JavaScript application. **Third**, as SINATRA requires minimal changes to the target page, the proxy performs those changes automatically for all the pages accessed by the browser. The proxy must also be able to intercept HTTPS traffic.

There is an off-the-shelf proxy that meets all the requirements: `mitmproxy`[12] can intercept HTTPS traffic (through an extra root certificate), is highly configurable with custom Python code, and can redirect traffic from one connection to another. We validated the feasibility of using `mitmproxy` for SINATRA’s purposes through a series of small throw-away prototypes, and we report that `mitmproxy` can indeed be used with SINATRA. However, for the sake of implementation and experimentation simplicity, our current implementation does not use a proxy, as we perform all the changes manually on static HTML pages that do not issue XML HTTP requests.

3.2 DSU with Sinatra

SINATRA performs updates over 3 different phases, as shown in Figure 4.

Phase 1 executes for the vast majority of the time, when no update is taking place. This is the *single-version phase*, which runs a single browser version in isolation. In this phase, SINATRA intercepts all JavaScript events and sends them to a *Coordinator* process, which simply keeps them in memory until the events are needed by later phases. Of course, the coordinator needs to have enough memory to store all the events generated on the browser due to user interaction. Sections 6.3 and 6.4 show that SINATRA’s memory requirements are modest, well within the capabilities of modern computers. In instances where a webpage executes network requests in the background, the observed memory requirements remain modest even with a long-running session. A webpage left open will also record any network activity that takes place, such as updates to a live Twitter feed. We describe such an experiment on Section 6.6, which requires 36MB of event storage for a 14h run.

Phase 2 is when updates start, during which the user launches the new browser version. For each page, SINATRA sends all the events from the coordinator and to the new version. Note that SINATRA transfers the state in the background, which allows users to continue to interact with the old browser. Events generated by user interaction during Phase 2 are simply added to the end of the list of events that the new browser needs to process. Section 6.5.1 shows that SINATRA takes, at most, 1.5 seconds in Phase 2.

Phase 2* is optional, and starts when the new browser has processed all the events in the coordinator’s log. During Phase 2*, SINATRA performs MVX between the old browser and the new browser. Phase 2* allows to validate whether an update was successful, by comparing each page on the leader with its version on the follower (e.g., matching their DOM tree). If the pages do not match, Phase 2* allows to stop an update that results in loss of data without any disruption, simply by terminating the follower and reverting to Phase 1.

```

1: let originalHandler = element.onclick;
2: if (originalHandler) {
3:   closure = function (ev) { id = element.sinatra_id;
4:     sendToCoordinator("onclick", id, ev);
5:     originalHandler.call(element, ev); }
6:   registerHandler(element, "onclick", originalHandler);
7:   element.onclick = closure; }

```

■ **Figure 5** Interception of a DOM0 event in the leader. Functions `sendToCoordinator` and `registerHandler` shown in Figure 8 and discussed in Section 3.5.

Validating the pages can be done manually (asking the user if all pages look the same to the user) or automatically (fuzzy matching the DOM contents, or using computer vision algorithms to find differences in screenshots of the rendered pages [25, 67]). Our prototype supports Phase 2* for benchmarking convenience, allowing us to measure the time to transfer logs and to swap roles with great accuracy.

Phase 3 effectively finishes the update by switching the browser exposed to the user. SINATRA *demotes* the old browser version, which becomes the follower, and *promotes* the new browser version, which becomes the leader. At this point, SINATRA can terminate the old browser and start a new *Phase 1*, as the browser was successfully updated with zero downtime and without losing any state. Phase 3 causes the only user-noticeable pause, which we measured in Section 6.5.2 as less than 10ms. For evaluation convenience, our prototype keeps executing the old browser as the follower until the user terminates the old browser.

3.3 Intercepting Events

SINATRA establishes the foundation for MVX and browser updates by intercepting events and sending them from the leader to the follower (through the coordinator). This section explains how SINATRA captures browser events in pure JavaScript by intercepting handlers along with their parameters on the leader.

SINATRA intercepts events by replacing the original event handler with a special handler. This way, when a message causes the event loop to execute a handler, SINATRA's code executes instead, which allows SINATRA to intercept the event that triggered the handler together with the actual handler that is executing. Messages are generated by either DOM0 or DOM2 event listeners:

DOM0 events. DOM0 events can be registered in-line on the HTML page (e.g., Line 7 on Figure 2), and through JavaScript properties on the DOM elements (e.g., Line 12 on Figure 2).

Intercepting DOM0 events is straightforward, as these handlers can be listed/modified directly from DOM elements, simply by reading/writing the respective property, respectively (e.g., Lines 1 and 7 on Figure 5). However, there are two challenges with intercepting DOM0 events. **First**, DOM0 handlers are only present *after* the HTML page loads and executes all in-line scripts. SINATRA's initialization code runs precisely at the right time, just after the HTML page loads but before any handler can be triggered, as shown in Line 3 of Figure 2. At this time, a simple DOM traversal can intercept all inline DOM0 event handlers. **Second**, the page can change DOM0 events without SINATRA noticing. SINATRA uses a mutation listener [60] to register a closure that runs when properties of elements change.

Figure 5 shows how SINATRA uses to intercepts DOM0 events. For each DOM0 handler (Lines 1), SINATRA captures the original handler (Line 2) and replaces it with its own closure (Line 8) that captures the current DOM element – `element` – and the event – `ev`, sends them to the coordinator (Line 5), and runs the handler originally registered by the JavaScript application (Line 6). Note that SINATRA only installs DOM0 events when needed (Line 3).

```

1: const originalAddEventListener = HTMLElement.prototype.addEventListener;
2: HTMLElement.prototype.addEventListener = function(evType, evListener, u) {
3:   ownerId = this.sinatra_id;
4:   registerHandler(this, evType, evListener);
5:   let closure = function (ev) { id = ev.target.sinatra_id;
6:                               sendToCoordinator(evType, ownerId, id, ev);
7:                               evListener.call(this, ev, u); }
8:   originalAddEventListener.call(this, evType, closure, u); }

```

■ **Figure 6** Interception of DOM2 events on the leader. Functions `sendToCoordinator` and `registerHandler` shown in Figure 8 and discussed in Section 3.5.

DOM2 events. DOM2 events register handlers by calling method `addEventListener` on the target element (e.g., Lines 14–16 on Figure 2). Registering a DOM2 handler requires two arguments: (1) type of event (e.g., `change` for when the target text input box changes), and (2) the event handler itself, specified as a JavaScript closure. Unfortunately, it is not possible to list handlers installed via DOM2. Furthermore, DOM2 describes a complicated logic about how events “bubble” and call all registered event handlers by following the DOM tree and combining DOM0 and DOM2 events. We discuss how bubbling affects SINATRA in Section 4.2. SINATRA intercepts DOM2 events by replacing `[HTMLElement, HTMLDocument].prototype.addEventListener` with its own closure, shown in Figure 6. Note that it is not possible for the underlying JavaScript program to install a DOM2 handler before SINATRA installs its own because SINATRA installs the handler before any other code runs, shown in Line 3 of Figure 2 (scripts without `defer` are downloaded and executed immediately). From this point onwards, when the JavaScript application calls `addEventListener`, SINATRA’s code executes instead (Lines 3–8). To intercept DOM2 events, SINATRA installs its own closure using the original `addEventListener` function (Line 8). Then, events that trigger the handler execute SINATRA’s closure which starts by sending the event to the coordinator (Line 4) before calling the original handler that the JavaScript application registered (Line 5).

Dynamically created elements. Dynamically created elements can also have event listeners, even before being added to the DOM tree. DOM2 listeners are automatically instrumented, as they use the prototype `HTMLElement` which SINATRA already instruments. SINATRA intercepts DOM0 events through a *Mutation Observer* [60] for new nodes added to the DOM tree, which SINATRA instruments as described above in this section.

Timers. Timers register a closure to execute after a specified time interval through functions `setTimeout` – a one-off event – and `setInterval` – a repeating event, as shown in Lines 3–5 of Figure 7. Of course, such timers are yet another source of synchronous non-determinism that SINATRA must handle. SINATRA uses an approach similar explained above in Section 3.3 and replaces functions `setTimeout` and `setInterval` with SINATRA’s own (Line 9). Then, when the underlying application registers a timer, SINATRA transparently intercepts those calls to register its own timer (Lines 13). When the timer expires, SINATRA intercepts the timer event and sends it to the coordinator before executing it (Line 12).

XML HTTP Requests (XHR). XML HTTP Requests (XHR) require an `XMLHttpRequest` object, which defines a number of properties with different roles: (1) hold the data obtained from the remote server (e.g., `status`, `responseText`), or (2) hold closures to be invoked with the XHR changes state (e.g., `onload`, `onreadystatechange`). SINATRA intercepts the function that creates such objects (i.e., `new XMLHttpRequest`) to return a proxy XHR object

26:10 Sinatra: Instantaneous Updates for Browsers Through Multi-Version eXecution

```
01: // Sample program
02: buttons = [] // A list of buttons
03: setInterval(30, function() { // Expensive computation
04:     r = Math.random(); button[r].enabled = true; });
05: setInterval(300, function() { buttons[...].enabled = false; });
06:
07: // Interception
08: let uniqueID = 0;
09: let realSetInterval = setInterval;
10: setInterval = function(origHandler, delay) {
11:     let myID = uniqueID++;
12:     registerInterval(delay, myID, origHandler);
13:     let closure = function() { intervalToCoordinator(myID); origHandler(); };
14:     return realSetInterval(closure, delay); }
```

■ **Figure 7** Sample JavaScript program that uses timeouts (Lines 1–5) and `Math.random` (Line 4); and how SINATRA intercepts timer events (Lines 8–14). Function `registerInterval` and `intervalToCoordinator` are explained Figure 8 and on Section 3.5.

that contains a real XHR object internally. Then, SINATRA defines the same properties as the real XHR object, through `Object.defineProperty`, and uses them to intercept how the JavaScript manipulates the proxy XHR object [14].

As with the DOM events described above, the leader issues XHR requests and sends the returned values to the follower. The follower does not issue any XHR request, it simply gets the data from the leader. When the leader executes a closure associated with XHR state change, it sends information to the follower to trigger the execution of the same closure with the same data. Note that leader and follower issue the same XHR requests by the same order, so SINATRA identifies each XHR request uniquely by the order in which they are issued.

3.4 Intercepting Asynchronous Non-Determinism

As described in Section 2.3, JavaScript programs can call functions that are non-deterministic. The most important such function is `Math.random`, which is used extensively by many JavaScript applications. Unfortunately, using the same approach described in Section 3.3 does not work due to the asynchronous nature of the call to such non-deterministic functions.

For instance, consider the example shown in Figure 7. In this example, there is a list of buttons (Line 2), all disabled. Every 30 seconds, the program performs an expensive computation (Line 3) and enables one button at random (Line 4). Every 5 minutes (300 seconds), the program disables all buttons again (Line 5).

Now consider the following implementation: SINATRA captures the 30 second event, sends it to the coordinator, then captures the execution of `Math.random`, and also sends it to the coordinator. This approach works if all the events are known in-advance (i.e., Phase 2 of Figure 4 and existing record-replay approaches). However, **this approach does not work for MVX** (i.e., Phases 2* and 3 of Figure 4). In this case, it is possible that the follower receives the timer event and reaches the call to `Math.random` before the leader, as the time required to perform the expensive computation may not match in both versions, and the follower may complete it before the leader. At this point, the follower does not know which number to return to match the leader. Making matters even worse, the follower cannot simply wait for the leader, because doing so in JavaScript’s event-loop model results in a deadlock, as explained in Section 2.3.

Functions that result in asynchronous non-determinism thus need special consideration. One way to deal with `Math.random` is to use the same seed for the underlying Pseudo-Random Number Generator (PRNG). Unfortunately, it is not possible to seed JavaScript’s PRNG.

An alternative is to replace calls to `Math.random` with a custom PRNG that can be seeded, on all variants, which may result in a lower quality source of randomness. Instead, SINATRA starts by generating a sequence of N random numbers when the page loads (i.e., in the `script` tag on Line 3 of Figure 2). Then, each call to `Math.random` consumes one number from the sequence. The leader replenishes the sequence by sending a fresh random number to the coordinator for each number consumed. This approach allows a fast follower to consume up to N random numbers asynchronously at its own pace, ensures all random numbers match between leader and follower, and provides a fresh supply of browser-grade randomness. This approach also simplifies detecting divergences between leader and follower, as we can check in the sequence of random numbers how far/behind one variant is. We found that a cautious value of $N = 100$ works well in practice.

Methods in the `Date` object also require special treatment. The leader starts by consulting the current date/time, saves it in a variable, and sends it to the follower. Then, when the JavaScript program attempts to consult the date, both leader and follower return the saved date, and increment it (e.g., by 100). As such, both versions agree on all dates generated. To ensure fresh and realistic dates/times, every so often, just before sending another message to the follower, the leader refreshes its saved date with the system's and adds date information to the message being sent.

3.5 Multi-Version Execution in JavaScript

So far, this document describes how to capture all the sources of non-determinism used by a JavaScript program on the leader browser. But this is only one half of the problem. To transfer the state between browsers, and to keep them synchronized after that, SINATRA needs to ensure that the follower browser sees exactly the same non-determinism (i.e., the same events in the same order on the same DOM elements).

Matching elements. SINATRA assigns IDs (monotonically increasing numbers) to each DOM element by adding a new property `sinatra_id`, traversing the initial DOM tree after the page is loaded, and then for each dynamically added element (described in Section 3.3). Given that SINATRA traverses the same DOM tree in a deterministic way, and executes `createElement` in the same order in both browsers, the same element always receives the same ID in both browsers.

SINATRA keeps a global structure with all the handlers registered, as shown in Figure 8 (Line 2). When registering events, SINATRA keeps a map for each element ID (Lines 6–9). The map associates event types (e.g., `onclick`) to the respective handler and the target element in which the event was registered (Line 9). We note that each browser keeps references to its own handler and element.

The leader sends events to the coordinator via function `sendToCoordinator`, which serializes the event as discussed in Section 4.2. The follower receives deserialized events from the coordinator via function `receiveFromCoordinator`, which consults the global structure to get the target element (Line 19) and the handler (Line 20) registered for the current event being triggered. Then, the follower calls the handler directly, setting the receiver as the target element (Line 21).

Ensuring the same causal ordering. Given JavaScript's event-driven model, is possible to violate causal ordering in the follower where events are delivered targeting elements that do not exist (yet). For instance, consider a page with one button (1) that, when pressed, creates another button dynamically (2). Consider also an execution in which the user presses buttons

26:12 Sinatra: Instantaneous Updates for Browsers Through Multi-Version eXecution

```
01: // Globals in both leader and follower
02: let globalHandlerTable = {};
03: let intervalHandlerTable = {};
04:
05: // Executed by both leader and follower
06: function registerHandler(el, eventType, handler) {
07:     let id = el.getAttribute("sinatra_id");
08:     if (!globalHandlerTable[id]) globalHandlerTable[id] = {};
09:     globalHandlerTable[id][eventType] = {"handler": handler, "target": el}; }
10: function registerInterval(delay, id, handler) {
11:     intervalHandlerTable[id] = { "delay": delay, "handler": handler }; }
12:
13: // Leader calls:
14: function sendToCoordinator(eventType, elementID, eventObject) { ... }
15: function intervalToCoordinator(id) { ... }
16:
17: // Follower calls:
18: function receiveFromCoordinator(eventType, elementID, eventObject) {
19:     let targetElement = globalHandlerTable[elementID][eventType]["target"];
20:     let handler        = globalHandlerTable[elementID][eventType]["handler"];
21:     handler.call(targetElement, eventObject); }
22: function intervalFromCoordinator(id) { intervalHandlerTable[id]["handler"](); }
```

■ **Figure 8** Matching events and timers to handlers and elements in both leader and follower browsers.

(1) and (2). On the follower, it is possible that both button presses are added to the message queue (depicted in Figure 1). Creating the hypothetical button (2) generates another event, which is then added to the message queue *after* the event for pressing that same button (2). This execution violates causal ordering and results in pressing a non-existing button.

To keep causal order consistent between variants, SINATRA uses the latest `sinatra_id` as a logical clock sent with each event from the leader to the follower. Adding new DOM elements results in generating a new `sinatra_id`. When receiving an event, the follower checks the event's `sinatra_id` against the follower's latest `sinatra_id`. If the values do not match, the follower simply postpones processing the event by adding it to the end of the queue with `timeout(0)`, as explained in Figure 1. SINATRA does the same for all events, including XHR requests.

Matching timers. The follower never registers timers and XML HTTP Requests with the browser. Instead, the follower executes the closures registered with each handler in the order that the leader issues them through the coordinator. However, this creates a problem: How can the follower distinguish between many different closures? For instance, consider the example shown in Figure 7. This example installs two closures associated with different timeouts, one in Line 3 and another in Line 5. When one of these expires and the leader executes it, how can the follower know which to execute?

SINATRA uses a unique ID to differentiate each closure registered with a timer (Lines 8 and 11 in Figure 7). Given that SINATRA ensures that the follower executes the same event handlers by the same order as the leader, the IDs always match between variants. Both variants then keep a table from IDs to closures and delays (Line 3 and Lines 10–11 in Figure 8). When sending an event about an expired timer, the leader sends the ID of the closure associated with the timer (Line 12 in Figure 7). The follower then uses the ID to address its table, get the correct closure, and execute it (Line 22 in Figure 8).

Promotion/demotion. When the roles of the browsers switch (Phase 3 in Figure 4), SINATRA uses the information kept on the global structure (Line 2 on Figure 8) to install all event handlers on the respective DOM elements on the promoted browser, and removes them on the demoted browser. SINATRA also cancels all the timers on the demoted browser, and installs them on the promoted browser (with the original timeout value). Because SINATRA does not track how much time passed since each timer was installed, this approach may cause timers to expire after longer than needed ($2\times$ in the worst case). However, this is correct as timers in JavaScript guarantee only a minimum amount of time to wait before triggering the associated closure. Pending XHR at the time of promotion cause the leader to postpone the swap until the pending requests are completed. During this time, all new requests are deferred until the follower is fully promoted and eligible to initiate the requests. This leads to user observable pauses until the pending XHR requests are resolved, which Section 6.6 measures as 86ms on a realistic workload.

Read-only Follower. In the context of MVX we now have two browsers, as shown in Figure 4. The user interacts with a *leader* browser, which sends all the non-determinism to the coordinator process. Then, a *follower* browser receives the same non-determinism from the coordinator. This way, SINATRA ensures both leader and follower are always synchronized.

Users can inspect the state of the follower browser, but they cannot modify it because the follower intercepts all the handlers as described in Section 3.3, but does not install any event handlers with the browser. Instead, the follower registers events just with SINATRA (i.e., Line 7 in Figure 5 sets `onclick` to `null`, Line 7 in Figure 6 and Line 13 in Figure 7 are omitted). Also, this approach ensures that the follower executes timer handlers in sync with the leader, running them only when the leader sends the respective events.

4 Implementation

In this section, we describe the implementation details of SINATRA. SINATRA is implemented in pure JavaScript, totaling 2013 lines of code. The web APIs leveraged by SINATRA to intercept user and system generated events are compatible with the most recent versions popular browsers, such as Google Chrome, Mozilla Firefox, Apple Safari, and others. SINATRA works out of the box for most browsers, without requiring external packages, tools, or plugins.

4.1 Coordinator and Protocol

The coordinator process enables communication between both browsers, which is at the core of SINATRA's approach to MVX, and keeps a log of JavaScript events during Phase 1, as shown in Figure 4. We implemented the coordinator process using *node.js* [3], so it executes in its own separate (headless) process without a browser. We use the *SocketIO* [4] JavaScript library to enable bi-directional communication between the coordinator and each browser.

The initialization protocol for SINATRA is quite simple. First, the coordinator should be executing before any browser is launched. On browser launch, SINATRA starts by connecting to the coordinator using a pre-configured address and port, and sends a message. The coordinator replies with the role of this browser, which is *leader* for the first browser and *follower* for the second.

Upon learning its role, a leader browser generates the list of random numbers mentioned in Section 3.4, sends it to the coordinator, and starts sending all events from that point on. A follower browser, conversely, waits for the coordinator to send the list of random

numbers, followed by the events that were kept during the leader's execution. At this point, the coordinator informs the follower that it is synchronized, and MVX starts. During MVX, the coordinator sends each event, received from the leader, to the follower as it is received.

Given that communication is bidirectional, the coordinator does not have to establish new channels when promoting the follower/demoting the leader. Instead, each browser simply changes roles and execution continues in MVX, but in the opposite direction. The promotion/demotion event starts from the leader (outdated browser), when the user presses a button that SINATRA injects at the top of the page. Together with a special promotion/demotion message, SINATRA also sends the list of pending timers that were cancelled and their timeouts, as described in Section 3.5.

4.2 Serializing/Deserializing Events and Bubbling

When the underlying JavaScript program executes an event handler on the leader, SINATRA's code is first called with the event. First, SINATRA gets the **name** of the event, (e.g., defined as argument `evType` in Line 2 of Figure 6). Second, SINATRA gets the **ID of the target element** – the element that triggered the change (e.g., defined as argument `element.sinatra_id` in Line 3 of Figure 6). As explained in Section 3.5, SINATRA ensures that all elements have a unique ID. Finally, SINATRA creates a JavaScript object to hold a copy of the event object, and populates it with **all the fields in the event object**, which include the coordinates of mouse events, which key was pressed that triggered the event, and other relevant data.

At this point, SINATRA can send the JavaScript object to the coordinator. The SocketIO implementation automatically turns the JavaScript object into its JSON representation [15] through function `JSON.stringify` on send, and back into a JavaScript object using function `JSON.parse`. The coordinator simply keeps a list of tuples (name, element ID, event) received from the leader. Sending this list to the follower, when it becomes available, requires another round of serializing to JSON by the coordinator, and deserializing back into JavaScript objects by the follower.

An important note is that SINATRA feeds the deserialized event directly into each handler in the follower, as shown in Line 21 of Figure 8. SINATRA does not create/trigger a new synthetic browser event, as some record-replay systems for JavaScript do through `DOMnode.fireEvent` [39]. This design decision simplifies handling *event bubbling*, when many handlers trigger for a single event (e.g., when a child DOM element has a different handler for the same event as its parent). Instead, SINATRA simply captures the order in which the leader executes the event handlers, and their respective targets; and then calls the same handlers by the same order in the follower. The alternative of creating synthetic events has well-known corner-cases that require special consideration. Furthermore, SINATRA can handle browser updates that change the bubbling behavior.

4.3 Stateful DOM Elements and Text Selection

DOM elements, such as radio buttons, check boxes, and text boxes, keep internal state. For instance, when the user selects a check box, the state of that check box changes (it is now selected). Updating the state does not execute any JavaScript handler, which means that SINATRA cannot intercept it directly. Fortunately, there are only a limited number of such elements, and SINATRA handles them as a special case by installing its own event handler associated with the **change** event, even when there is no application handler. The event handler simply captures the updated state of the DOM element, which allows the follower to remain synchronized with the execution on the leader.

Another source of state, and non-determinism, is the text selected by the user. JavaScript can access and manipulate the current text selection, based on a range of characters on a text element (start and end). To detect when the text selection changes, the leader listens for left mouse button releases, and **SHIFT** key releases. At that point, SINATRA can obtain the current text selection (if any), create a JavaScript object that captures the start and span of the selection, and send it to the coordinator. On the follower side, SINATRA uses the information received to select the same text.

5 Practical Considerations and Limitations

The main design goal of SINATRA is to remove all barriers to instantaneous and stateful browser updates, so that users can enjoy automatic browser updates without even noticing them. Another important design goal is to be applicable to all browsers by targeting JavaScript's execution model. This design choice leaves out of scope state maintained inside the browser itself, such as Websockets. For instance, the recent WebRTC standard allows for real-time audio-visual communication, started from JavaScript but implemented inside the browser [63]. Of course, by its design goals, SINATRA does not support such features implemented internally by browsers.

We argue that SINATRA is a practical approach in its current form. SINATRA deals with the browser state that is the most complex and hard to migrate: the JavaScript engine. State-of-the-art DSU approaches excel at dealing with the remaining state inside the browser [24, 36], which would require browser modifications. Perhaps browser vendors can provide an API to migrate open connections and other browser state. Finding such state, and how to migrate it, is exciting future work that is out-of-scope for SINATRA.

Still, SINATRA can deal with pages that hold internal browser state in three possible ways. First, simply reload them. In the WebRTC example, this means that the video-conference connection would drop and reconnect, which is a relatively common event that users tolerate. Second, wait until all such pages are closed and then update the browser. Third, list the unsupported pages and ask users if they accept reloading them.

The current prototype of SINATRA requires two external components: the proxy and the coordinator. We believe that these components can be implemented as plugins to popular browsers [40, 22], and present them here as separate components to highlight the fact that SINATRA works on unmodified browsers.

Other limitations. The main design goal of SINATRA is to allow instantaneous and stateful browser updates. As such, we designed SINATRA under the assumption that only one user interacts with each JavaScript program, and that each JavaScript program does not execute for a long time. All these assumptions break for server-side JavaScript applications written in node.js [3]: many users interact with each JavaScript program, and each program executes for a long time. Even though SINATRA can be applied to such programs, to update the node.js virtual machine, this is not feasible, as such applications handler numerous events within a short time span and result in very large log files. This is outside of the scope of SINATRA.

The current version of SINATRA does not handle persistent state created through `Window.localStorage` [62]. Our evaluation ensures that the persistent state is empty before each run. Supporting persistent state is straightforward: `function Object.keys(localStorage)` can iterate over all the persistent state at the start of execution, and SINATRA can send that to the follower to ensure the same initial persistent state.

SINATRA does not support the Web Workers (WW) API [61], which introduces multi-threading. However, each WW thread executes its own event loop (Figure 1 shows one event loop, each WW has its own event loop), and WWs can only communicate through sending/receiving messages or through a shared `ArrayBuffer` object. Supporting WW requires capturing the total-order of messages sent between different threads, which can be accomplished through Lamport clocks [33, 27, 43, 58]. We plan to add support for WW in future versions of SINATRA.

6 Experimental Evaluation

In this section, we evaluate the feasibility of using SINATRA to deploy browser updates in practice, by measuring the overhead it introduces in terms of perceived latency by the user, and extra memory needed by the user’s computer. We also evaluate SINATRA’s performance as an MVX tool to enable future research. In that regard, we pose the following *research questions (RQs)*:

- **RQ1:** Is the latency added by SINATRA noticeable by the user?
- **RQ2:** What is the average size of the log that SINATRA keeps?
- **RQ3:** How does the size of the log that SINATRA keeps grow with user interaction?
- **RQ4:** How long does SINATRA take to perform a browser update?
- **RQ5:** How much resources does SINATRA require to support XHR on realistic pages?
- **RQ6:** What is the latency when SINATRA is used as a JavaScript MVX system?
- **RQ7:** Can SINATRA be realistically used with modern pages that use complex JavaScript?

We used two versions of two popular internet browsers, Mozilla Firefox versions 82.0 and 83.0, and Google Chrome versions 88.0.4323.150 and 89.0.4389.72. Unless when using updates, both leader and follower used the lowest version of each browser. The experimental evaluation took place in a modern desktop computer running Ubuntu Linux 20.04 LTS 64bit, with an Intel(R) Core(TM) i7-9700K CPU 3.60GHz and 32GB of RAM.

6.1 Applications and Workloads

We evaluated SINATRA with the 4 JavaScript applications (describe below). Each application requires user interaction, using the keyboard and/or mouse. We automated such interaction using the tool `Atbswp` [2] to record mouse and keyboard interactions – *workloads* – for each application, and then replay them. `Atbswp` records mouse and keyboard interactions and writes an executable Python script that replays those events using the library `pyautogui` [56]. We now describe each program, and the workload we used:

nicEdit [31] uses JavaScript to add a rich-text editing toolbar to an HTML `div` element. The toolbar applies styles (e.g., bold, italic, underline, font, color, size) to the text selected via the `document.execCommand` JavaScript API [29]. `nicEdit` creates the toolbar dynamically, using `document.createElement` to generate buttons and custom screens (e.g., to input the URL and text of an hyperlink), and attaches DOM0 event listeners to each generated element (i.e., buttons on the toolbar). `nicEdit` also creates a `textarea` element dynamically, where the user can input text.

The workload for `nicEdit` is representative of a user editing text. It starts with a pre-generated text, selects sections of text, and edits each in a different way: making the text bold, italic; changing the font size, font family (Arial, Helvetica, etc), and font format (heading and paragraph). The workload also changes the indentation of a paragraph, increasing it twice and then returning the paragraph back to its original indentation.

DOMTRIS [52] is a dynamic-HTML based Tetris game that uses JavaScript to implement the game mechanics: generate random pieces of different sizes, shapes, and colors; intercept user input; and schedule the movement of each piece inside the Tetris board. DOMTRIS uses `setTimeout` to schedule the downwards movement of the current piece, and `Math.random` to pick the next piece from the available set of pieces. The player interacts with DOMTRIS solely via the directional arrows on the keyboard, intercepted via DOM2 event listeners. Each piece is created dynamically with `document.createElement`.

We automated a Tetris game that drops each piece (without rotating it) on the center, extreme left, and extreme right of the board. As time goes on, the board fills along the center and edges until there is no space left, at which point the game ends. We understand that this is not how people play Tetris, but we cannot use `Atbswp` to automate a valid Tetris game because DOMTRIS uses a different random seed for every game. Even though SINATRA could generate a fixed sequence of random values to ensure deterministic replays, doing so would not show that SINATRA keeps the random values synchronized between browsers.

Painter [49] allows users to draw pictures using the mouse, with various colors and tools (free-hand brushes, lines, rectangles and circles). The user interacts with Painter using only the mouse over 3 HTML5 `canvas` elements [64]: (1) the tool set, (2) the drawing area, and (3) the color and line-width picker. Painter tracks the mouse position and button click/drag using DOM2 events, and reacts to different tool and color selections using DOM0 events. Painter generates a large number of events as it tracks the mouse movements at all times.

Our workload draws a tic-tac-toe board with the brush and line tools, then draws different shapes of different colors inside the board. This requires selecting different tools, colors, and brush strokes; effectively interacting with all parts of Painter. Note that `Atbswp` records the mouse with coarse precision between mouse clicks, which results in a low fidelity replay. For instance, when dragging the mouse along a line, `Atbswp` only captures the mouse position on the start of the line when the mouse button is pressed, one or two positions along the line, and the final position when the mouse button is released. We edited the generated Python script to ensure that the recording replays mouse movements on a pixel-by-pixel basis, to ensure high fidelity and accurate event counts. Unfortunately, due to `pyautogui`'s low performance when replaying a large number of mouse movements, the Painter workload takes minutes to execute what took us seconds to draw.

Color Game [28] is a game that tests reaction time via the Stroop Effect [54] (delay in reaction time between congruent and incongruent stimuli). The game shows players one color name, and requires players to press the button with the same name (out of 4 buttons), but with a different background color (e.g., press the red button with text “Blue” when the game specifies the color “Blue”). The game keeps track of the score (+5 for each correct click, -3 otherwise) during a 30 second round. Color Game is a complex application due to its use of the **Angular JS framework [21]**. Internally, Angular uses `document.createElement`, a combination of DOM0 and DOM2 event handlers, `setTimeout`, `setInterval`, and `Math.random`.

The workload consists of one run of the game (30 seconds), clicking each of the four color buttons in arbitrary order after every one second until the game ends.

Google and Twitter are popular pages representative of realistic workloads. The Google search page uses the **JsAction framework [23]** and Twitter uses the **React framework [38]**. The workload simply allows each page to load and then we interact with each manually.

■ **Table 1** Time required to run event handlers, average of 5 runs with standard deviation.

Program	Browser	Vanilla	Sinatra	Overhead	
				Relative	Absolute
nicEdit	Firefox	$2.600ms \pm 0.144$	$4.496ms \pm 0.371$	1.729×	+1.896ms
	Chrome	$3.779ms \pm 0.265$	$4.547ms \pm 0.208$	1.203×	+0.768ms
Painter	Firefox	$0.102ms \pm 0.018$	$1.570ms \pm 0.055$	15.361×	+1.468ms
	Chrome	$0.114ms \pm 0.004$	$0.982ms \pm 0.080$	8.639×	+0.869ms
DOMTRIS	Firefox	$0.495ms \pm 0.101$	$1.497ms \pm 0.094$	3.025×	+1.002ms
	Chrome	$0.250ms \pm 0.021$	$0.757ms \pm 0.027$	3.025×	+0.507ms
Color Game	Firefox	$1.457ms \pm 0.084$	$1.797ms \pm 0.041$	1.233×	+0.340ms
	Chrome	$0.663ms \pm 0.035$	$0.761ms \pm 0.026$	1.148×	+0.098ms

6.2 Sinatra Latency

To measure the extra latency added by SINATRA to each event on the leader, we compared the execution of each program without SINATRA (*vanilla*) and with SINATRA. The vanilla version measures the time taken to execute each original event handler during the workload. The SINATRA version measures the time taken to also execute SINATRA’s logic together with the original event handler. We measure the runtime of each event handler triggered during the workload, and report the average time among all the event handler executions observed. Table 1 shows the results.

This experiment highlights the extra latency that SINATRA adds to each event. Table 1 shows that SINATRA increases the latency by a maximum of +1.896 (nicEdit on Firefox), from 2.6ms to 4.496ms. The maximum increase in relative terms is for Painter on Firefox, at 15.361×, which translates to a low absolute increase of +1.468ms, from 0.102ms to 1.570ms. The results answer **RQ1: Users cannot notice the extra latency introduced by Sinatra.**

6.3 Log sizes

SINATRA spends the vast majority of the time executing in single-leader mode, as described in Section 3.2. In this mode, SINATRA stores a log in the coordinator with all the events and handlers that the (single) leader executed. In this experiment, we executed the workload for each application in single-leader mode to measure the size of the log on the coordinator, in number of events and size of the log. Table 2 shows the results.

We can see that the number of events varies widely between different experiments. nicEdit has the smallest number of events, as styling text results in a low number of button clicks and text selections. Color Game has twice as many events as DOMTRIS, which involve user input, timers expiring, and random number generation. Finally, as expected, Painter generates the largest number of events due to its fine-grained tracking of mouse events. In terms of absolute log size, we can see that all logs are below 5.4MB. The results of this experiment allow to answer **RQ2: Sinatra requires a modest amount of memory to store the log, below 5.4MB per page.** This result shows the practical applicability of SINATRA, given that average modern computers measure memory in tens of GB.

■ **Table 2** Log sizes in single-leader mode, average of 5 runs with standard deviation.

Program	Browser	# of events	Log size (bytes)
nicEdit	Firefox	278 ± 0.0	142,717 ± 0
	Chrome	286 ± 0.0	122,197 ± 4
Painter	Firefox	2,077 ± 2.0	5,305,731 ± 5,056
	Chrome	2,049 ± 1.3	4,201,712 ± 2,708
DOMTRIS	Firefox	683 ± 0.4	980,094 ± 46
	Chrome	682 ± 0.0	851,230 ± 9
Color Game	Firefox	1,030 ± 8.7	1,533,970 ± 25,632
	Chrome	744 ± 4.9	949,111 ± 12,141

6.4 Sinatra scalability

User interactions with websites may differ in length of time and number of events triggered. To measure how SINATRA behaves with different lengths of interaction, we designed an experiment that uses 3 workloads for each application – small, medium, and large – modified as follows.

nicEdit. Repeat the experiment N times, each time performing the same various text changes that have been described previously. **Small:** $N = 2$. **Medium:** $N = 4$. **Large:** $N = 6$.

DOMTRIS. Move Tetris pieces to one, two, or three sides of the board. **Small:** Left side only. **Medium:** Left and right sides. **Large:** Left, right, and center.

Painter. Repeat the drawing N times, pressing the “Clear” button (which clears the canvas) in between. **Small:** $N = 2$. **Medium:** $N = 4$. **Large:** $N = 6$.

Color Game. Play a game N times, restarting it at the end of each 30 second run by pressing the “Restart” button. **Small:** $N = 1$. **Medium:** $N = 3$. **Large:** $N = 5$.

We repeated the experiment for each size, in single-leader mode, and we measured the duration (seconds), the total number of events in the log (thousands), the size of the log (MB), and the bandwidth needed to send all the events (KB/s). The bandwidth is computed from the duration and the size of the log, and intended to show how much the log grows as the user keeps interacting with a page over time. Table 3 shows the results.

For most experiments, the bandwidth remains roughly constant even as the length of interaction increases, which is to be expected. Color Game is the notable exception, in which the bandwidth increases with the length (and intensity) of user interaction. We believe this is due to internal AngularJS behavior that: (1) never cancels timers with the browser, simply executes a test to return from cancelled timers, which results in more timers expiring as the game is played again and again; and (2) installs `hover` handlers for all elements, which call `Math.random` and result in more handlers executing as the experiment moves the mouse to the “Replay” button and back to the playing area. Over time, this results in Color Game generating the largest log files, which is understandable as Color Game is a game that requires intense user interaction. Painter generates large log files because it targets all the mouse movements with a fine level of detail (pixel by pixel, as discussed above). Overall, the bandwidth stays under $253KB/s$, which is acceptable.

We note that the original Painter interaction took about 20sec, the runtimes shown in Table 3 are artificially inflated by the slow speed of `pyautogui`. The original bandwidth would be $1MB/s$, which is acceptable for applications that track the mouse with fine detail.

■ **Table 3** Duration, number of events, log size, and bandwidth needed for growing workloads. **S** means *small*, **M** means *medium*, and **L** means *large*. **FF** means *Firefox*, and **Chr** means *Chrome*. Average of 5 runs.

		nicEdit			DOMTRIS			Painter			Color Game		
		S	M	L	S	M	L	S	M	L	S	M	L
Duration (sec)	FF	51	85	119	53	73	91	466	914	1,364	42	122	202
	Chr	51	85	119	53	73	90	464	921	1,366	43	122	202
# of evts ×1000	FF	0.46	0.82	1.18	0.47	0.64	0.63	4.12	8.36	12.14	1.81	9.24	22.53
	Chr	0.48	0.86	1.24	0.47	0.47	0.66	4.09	8.17	12.24	1.76	8.73	21.01
Size (MB)	FF	0.26	0.48	0.71	0.67	0.94	1.16	10.56	21.46	31.15	3.05	19.51	50.97
	Chr	0.22	0.42	0.61	0.58	0.83	1.00	8.40	16.78	25.16	2.59	16.15	41.25
Bandwidth (KB/s)	FF	5.0	5.7	6.0	12.7	12.8	12.8	22.7	23.5	22.8	71.7	159.6	252.3
	Chr	4.3	4.9	5.1	11.0	11.4	11.0	18.1	18.2	18.4	60.9	132.0	204.1

The results of this experiment provide an answer to **RQ3: Sinatra logs grow at a rate of 253KB/s as the user interacts with a page.** This result is acceptable, as mouse-based user interactions are short and the bandwidth is not a bottle-neck for inter-process communication. We note that the result is much smaller for all the other cases.

6.5 Browser updates with Sinatra

SINATRA can be used to deploy a browser update without incurring any loss of (JavaScript) state on the pages opened by the running browser. Such updates involve: (1) transferring the JavaScript state to the updated browser, running as follower, by processing the log it receives from the coordinator; and (2) promoting the follower to be the new leader. This section describes two experiments, one for each of the steps.

6.5.1 Log processing time

This experiment measures the time that the updated browser, running as follower, takes to process all the events in the log sent by the coordinator. We executed the workload for each program (to completion), and then launched the new browser as a follower. On the follower, we took two measurements: (1) the time taken since the page is loaded until the follower is up-to-date with the leader, and (2) the time taken just processing the event log sent by the coordinator. Note that (1) includes all the SINATRA initialization logic plus (2). Columns “Process log” and “Start executing” of Table 4 show the results for (2) and (1), respectively.

We can see that processing the log of events is an important portion of the overall time required to start a follower. Most cases take under $338ms$, except Color Game. Color Game takes much longer to process the events in both browsers. We believe this is due to the underlying Angular.JS initializing a large number of libraries it uses as dependencies. Color Game takes longer on Firefox than on Chrome, which we believe is due to Chrome’s higher performance when executing Angular.JS code.

6.5.2 Time taken to promote follower

This experiment measures how long it takes to promote the follower to be the new leader (and demote the leader to become a follower) once the follower is up-to-date (i.e., after the follower processes all events sent by the coordinator). The experiment uses two browsers: B_1 as the initial leader, and B_2 as the initial follower. We execute half the workload by

■ **Table 4** Time from launching a follower until its state is up-to-date, time to promote, and round-trip time (RTT) between the leader triggering an event and receiving an acknowledgement from the follower for that event. Average of 5 runs with standard deviation.

Program	Browser	Process log (ms)	Start executing (ms)	Promote (ms)	Round-trip-time (ms)
nicEdit	Firefox	75 ± 6	168 ± 7	8.4 ± 2.3	13.56 ± 0.51
	Chrome	138 ± 7	277 ± 43	8.0 ± 2.0	19.77 ± 2.50
Painter	Firefox	338 ± 56	658 ± 111	6.7 ± 2.4	4.08 ± 0.05
	Chrome	491 ± 23	691 ± 13	4.6 ± 1.6	6.47 ± 1.51
DOMTRIS	Firefox	147 ± 19	420 ± 50	6.6 ± 2.2	32.61 ± 9.10
	Chrome	248 ± 87	438 ± 43	3.8 ± 0.8	26.71 ± 0.09
Color Game	Firefox	1,067 ± 103	1,435 ± 129	7.2 ± 2.9	19.03 ± 1.29
	Chrome	704 ± 52	1,180 ± 59	4.0 ± 2.0	17.15 ± 0.49

■ **Table 5** Latency observed by SINATRA when contacting a server via XHR with a fixed latency, and time require to change roles between variants. Average of 5 runs with standard deviation.

XHR Latency (ms)	Browser	Observed Latency (ms)	Promote (ms)
0	Firefox	7.12 ± 0.34	12.75 ± 1.30
	Chrome	6.60 ± 0.34	11.00 ± 3.08
50	Firefox	56.42 ± 0.42	52.50 ± 2.69
	Chrome	57.27 ± 0.32	56.75 ± 5.07
100	Firefox	107.42 ± 1.13	102.50 ± 1.66
	Chrome	108.84 ± 1.14	111.00 ± 6.78
1000	Firefox	1,008.25 ± 0.92	1,006.00 ± 6.20
	Chrome	1,009.74 ± 2.18	1,012.25 ± 9.44

interacting with B_1 , then switch their roles, then finish the workload by interacting with B_2 . We checked visually that the experiment behaves as expected, and measure the time taken since switching the roles of each browser. Column “Promote” on Table 4 shows the results. We can see that all promotions happen under $10ms$.

6.5.3 Time to perform an update

Putting together Sections 6.5.1 and 6.5.2 allows us to estimate the minimum time required to perform an update. Even though it may take a follower browser as long as 1.435 seconds to synchronize its state with the leader, this process takes place in the background and does not cause the user to stop interacting with the (leader) browser. Then, once the follower’s state is up-to-date, the promote/demote process takes less than $10ms$, which humans perceive as instantaneous. These two experiments also allow us to answer **RQ4: Sinatra requires an imperceptible pause (10ms) to update a running browser, and requires less than 1.5 seconds to prepare that update in the background since launching the updated browser.**

6.6 XML HTTP Request support

We evaluate SINATRA’s support for XML HTTP Request (XHR) with two experiments. First, we designed an experiment in which a leader and a follower perform 100 XHR requests in sequence to a local server that waits a certain amount of time (0ms, 50ms, 100ms, and 1s)

before sending back 100 bytes. On the 50th request, we swap the roles immediately after performing an XHR request, to force SINATRA to postpone the role swap as described in Section 3.3. We measure two things: (1) the latency observed by the leader, and (2) the time required to swap the roles. The results, presented on Table 5, show that SINATRA introduces little extra latency on top of the maximum XHR latency observed. Note that a latency of 100ms is not noticeable by the user.

In our second experiment, we captured all the XHR traffic during a period of 14h on a page that receive very frequent updates – the twitter feeds the local traffic and weather channel¹ – on both Chrome and Firefox. Then, we replay those requests using SINATRA and check the total log size needed. In this experiment, we measured an average number of requests of 5,676 for (1), 27,022 for (2); and an average log size under 6MB for (1), and under 36MB for (2). The overall average latency we observed was 86ms. These experiments allow us to answer **RQ5: Sinatra supports realistic XHR with modest storage requirements (under 36MB/14h), and introduces an imperceptible pause due to pending XHR (under 100ms).**

6.7 Using Sinatra as an MVX system

At its core, SINATRA is an MVX system targeting JavaScript. In this role, we are interested in measuring the latency between an event being triggered on the leader, and that same event being visible on the follower. We designed an experiment that measures the *Round-Trip Time (RTT)* of each event by sending an acknowledgement from the follower, for each event received, back to the leader, through the coordinator. The RTT provides a reasonable estimate of the leader-follower latency. This experiment runs the workloads for all the applications while measuring the RTT. Table 4 shows the results.

In all cases, the RTT is under $33ms$, which indicates a leader-follower latency of half the RTT, around $17ms$. The results from this experiment answer **RQ6: Used as an MVX system, Sinatra delivers events to the follower in 10ms after the leader.**

6.8 Using Sinatra on realistic webpages

To test whether SINATRA can be applied to pages that represent a realistic modern workload, we applied it to the Google search page and the Twitter home page (after login, showing a feed of tweets). We downloaded all the resources used by each page in advance, including XHR requests, to be able to observe the same execution reliably. We then modified the downloaded pages to add the required SINATRA headers, as explained in Section 3 and on Figure 2. We used Google Chrome for this experiment, and repeated each experiment 10 times. We measure the time to load each page by adding a closure with `timeout(0)` that readds itself, and measuring the time between each execution. Initially, the time between executions is high as the event-loop is busy loading the page. We measured an idle event-loop imposing a time between executions below $6ms$. When we observe 5 executions below $6ms$, we consider the page loaded.

Loading Google and Twitter, takes $267 \pm 61ms$ and $2891 \pm 321ms$, respectively. SINATRA processes a total of a total of 163 ± 2 events in $431 \pm 85ms$ and 4583 ± 37 events in $4234 \pm 451ms$, respectively. The overhead is 1.61 for Google, adding about $100ms$; and 1.46 for Twitter, adding about $1sec$.

¹ <https://twitter.com/TotalTrafficCHI>

Once loaded, the pages are fully functional and allow for user interaction. In MVX mode, the interaction on the leader is replicated on the follower without any noticeable delay, confirming that the results in previous experiments generalize to larger pages. Furthermore, we modified our local HTTP server to allow Google XHR traffic to go through, which enabled the search autocomplete feature as the user types to work correctly on the leader, being then replicated on the follower by SINATRA. The results from this experiment answer **RQ7: Sinatra can indeed be used on modern pages with sophisticated JavaScript that generate thousands of events with no loss of functionality and modest overhead.**

Threats to validity. Despite our best efforts, the evaluation in this document still has some threats to validity: (1) the websites we tested may not be representative of common websites, (2) the browsers/versions used may not be representative of popular browsers, (3) our results may be affected by bugs in SINATRA, and (4) using SINATRA on other websites may be affected by bugs in SINATRA.

7 Related Work

The problem of Dynamic Software Updating (DSU) has been a focus of past research, resulting in DSU systems for programs written in popular languages such as C [11, 24, 17, 18, 19] and Java [47, 55, 65, 66, 30, 45]. SINATRA differs from these systems in two important ways. First, SINATRA updates the *execution environment* and not the program running on that environment. For instance, DSU systems for Java do not support updating the underlying Java Virtual Machine while running the same program, which would be the closest to the goal of SINATRA. In fact, to the best of our knowledge, SINATRA is the first such DSU system outside of the Smalltalk community [20, 9] to target the execution environment specifically. Second, DSU systems typically require modifications to the programs being updated to support stopping the program in one version and resuming it in the next, and to express how to transform the state in the old program to an equivalent representation that is compatible with the updated code. In contrast, SINATRA works on unmodified closed-source commercial internet browsers. Instead of migrating the state directly, SINATRA launches the new browser as a separate process and migrates the state for each page individually. The only state kept outside of SINATRA is persistent HTTP connections, which SINATRA’s proxy keeps open during updates.

SINATRA uses Multi-Version eXecution (MVX) to synchronize the old and new versions of the updated browser. MVX has been used mostly in programs written in C/C++ by intercepting and synchronizing system-calls between processes. The main goal of MVX are: (1) to increase security [32, 13, 59], detecting divergences in potentially suspect processes; (2) to increase reliability [27, 8, 34, 37, 51], tolerating faults in one process by using the other processes; and (3) availability [26, 44, 48], by performing updates on a forked process and terminating it when updates fail, without any disruption. In fact, Mvedsua [44] is the most similar MVX system to SINATRA, given that it also combines MVX with DSU and allows users to build confidence on the validity of the update by executing both old and new versions for a period of time. However, Mvedsua targets C programs updated via Kitsune [24], intercepts system calls, requires modifications to the programs being updated and machine-parseable descriptions of the update-induced divergences. SINATRA requires much less developer effort, which can be fully automated using an HTTP proxy.

Record-Replay (RR) can be described as “offline Multi-Version eXecution”. It operates in two phases, typically using two different (automatically generated) programs. First, it records all non-determinism observed during an execution in a log file. Then, it uses that

log file to replay the same execution over the same program. By contrast, MVX records each non-deterministic datum in one process and replays it immediately on another process, thus keeping the state on both processes perfectly synchronized. MVX also needs to account for differences in execution speed that may result in a replayer overtaking the recorder and reaching a program point that requires non-determinism before that data is available. For this reason, RR approaches require the log to be complete before being able to replay it. Furthermore, RR approaches do not allow a replayer to become a recorder as they target a different problem: Accurate replication of bugs observed in production during development.

Techniques for RR target programs written in multiple popular programming languages: C [42], Java [7], and even web browsers [39, 10, 5, 25, 57]. Most techniques require a modified web browser. Dolos [10] and Jardis [5] use modified implementations of browser components (Webkit and ChakraCore) to record bugs in production and replay them in development and provide developers with time-travel debugging capabilities, respectively. Jardis focuses on node.js applications [3]. ReJS [57] also provides support for time-travelling debugging, but for any JavaScript code in general, through a modified version of Microsoft's ChakraCore JavaScript engine that performs heap checkpoints via a modified garbage-collector.

Working in pure JavaScript, Mugshot [39] is an RR system that demonstrates the feasibility of capturing all the needed non-determinism to ensure an accurate replay without the need for a modified web browser. Mugshot influenced the design of SINATRA by listing all the sources of non-determinism that need to be handled to capture all interactions between the environment and a JavaScript program executing in a browser. However, Mugshot relies on event listeners on the topmost DOM element (i.e., `window`) to intercept all events, and replays them through synthetic browser events (i.e., `DOMElement.fireEvent`). As a result, Mugshot has to deal with browser-specific behavior that impacts event bubbling and event ordering. SINATRA's approach of intercepting each handler individually avoids such complexity and naturally supports any browser without special handling. Similarly to SINATRA, X-Check [25] also works in pure JavaScript and works on different browsers (all other techniques require the same browser and version to replay the recorded logs). X-Check records logs on one browser and replays them on different browsers, with the goal of detecting cross-browser differences that developers can then replicate and address.

The closest system to SINATRA is Cocktail [67], an MVX system for web browsers with the goal of improving the security of internet browsers by feeding input to many different browsers and voting on the output. Cocktail can thus detect and defeat attacks that target a particular browser, or a particular browser version. Despite the very different goal, there are more important differences between SINATRA and Cocktail. First, Cocktail is implemented as a browser plugin and SINATRA is implemented in pure JavaScript. As such, SINATRA can be directly applied to any web browser, but Cocktail requires developer effort to write a new plugin for that browser. Also, Cocktail's plugin can intercept asynchronous non-determinism, such as calls to `Math.random`, and block until all browsers reach the same point. This is not possible in JavaScript's execution model, as described in Section 2.3. Second, Cocktail relies on an UI component to intercept mouse and keyboard events before they reach the browser. SINATRA captures the events at a finer level of detail, ensuring that all browsers execute the same JavaScript handlers by the same order, regardless of implementation-specific browser quirks that may show the same element on different positions in different websites. In fact, SINATRA can replicate the execution even if the leader and follower have different window dimensions, which is a limitation of Cocktail.

8 Conclusions and Future Work

This paper presented the design of SINATRA, a system that allows to update internet browsers without losing any state in the process. SINATRA works fully at the JavaScript level, using first-class function interception to keep track of all events, and then using MVX to perform updates on the new version of the browser while the old version keeps providing service. As a result, SINATRA works on popular, closed-source, commercial internet browsers such as Google Chrome. SINATRA requires a small amount of JavaScript source changes, performed to each page opened in the target browser. The changes required are easy to automate with a sophisticated internet proxy.

This paper also presented an extensive experimental evaluation, where SINATRA is applied to JavaScript applications with different combinations of features. When not performing an update, SINATRA imposes low overhead on the execution of event handlers (a max increase of 2.107ms). Also, the state that SINATRA keeps to support future updates grows at a modest rate of 10.8KB/s (at most) during intense user interaction. If a page remains open performing XML HTTP Request requests, SINATRA requires a modest 36MB of storage for a 14h run.

SINATRA can perform updates in short order, requiring just 1.5s (at most) to transfer the state from the old browser to the new browser. While SINATRA transfer the state, the user keeps interacting with the old browser. Then, to finish the update and allow the user to interact with the new browser version, SINATRA requires a very short pause in user interaction of less than 10ms, which is barely noticeable for most users.

Besides its role in browser updates, SINATRA doubles as an MVX tool for JavaScript applications. The experimental evaluation showed that SINATRA can keep two browsers synchronized, with an action on one browser taking effect on the other almost instantaneously, within 19ms.

In the modern internet age, an up-to-date internet browser is the first line of user defense. SINATRA dramatically lowers the barrier to deploy automatic and fully transparent browser updates by eliminating any data loss or service interruption associated with such updates. We strongly believe that SINATRA has the potential to improve the average user's safety by making disruptive browser updates a thing of the past.

Future Work. It is possible to use SINATRA to move a page from one browser to another (e.g., from Mozilla Firefox to Google Chrome). This feature can be valuable to security conscious users, who can switch browsers as a vulnerability is disclosed. We tested this feature of SINATRA informally to ensure it works, but did not evaluate it or develop it further.

SINATRA is OS and platform agnostic, and we plan to implement SINATRA on popular platforms (e.g., Microsoft Windows and Apple OSX) and apply SINATRA to the official browser in each platform (e.g., Microsoft Edge and Apple Safari).

References

- 1 Sinatra's github repository. <https://github.com/bitslab/sinatra>.
- 2 Automate the boring stuff with python. <https://github.com/RMPR/atbswp>, 2021. Accessed: 2021-04-14.
- 3 Node.js. <https://nodejs.org/en/>, 2021. Accessed: 2021-04-14.
- 4 Socket.io. <https://socket.io/>, 2021. Accessed: 2021-04-14.
- 5 Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. Time-travel debugging for javascript/node.js. In *FSE '16 Proceedings of the 2016 ACM International Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, September 2016. URL: <https://www.microsoft.com/en-us/research/publication/time-travel-debugging-javascriptnode-js/>.


- 6 Samuel Baxter, Rachit Nigam, Joe Gibbs Politz, Shriram Krishnamurthi, and Arjun Guha. Putting in all the stops: Execution control for javascript. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 30–45, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3192366.3192370.
- 7 Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicer: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 362–371. IEEE Press, 2013.
- 8 Emery Berger and Benjamin Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, volume 41, pages 158–168, January 2006. doi:10.1145/1133255.1134000.
- 9 Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009. URL: <http://pharobyexample.org>.
- 10 Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive record/replay for web application debugging. In *UIST 2013: Proceedings of the 26th ACM Symposium on User Interface Software and Technology*, pages 473–484, St. Andrews, UK, October 2013.
- 11 Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 271–281, USA, 2007. IEEE Computer Society. doi:10.1109/ICSE.2007.65.
- 12 Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 5.3]. URL: <https://mitmproxy.org/>.
- 13 Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USA, 2006. USENIX Association.
- 14 ECMA (European Association for Standardizing Information and Communication Systems). Standard ECMA-262 6th Edition – Section 19.1.2.4. <https://262.ecma-international.org/6.0/#sec-object.defineproperty>. Accessed: 2022-01-04.
- 15 ECMA International. Standard ECMA-404 – The JSON data interchange syntax. <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>, December 2017.
- 16 ECMA International. Standard ECMA-262 – ECMAScript(R) 2020 language specification. <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>, June 2020.
- 17 Cristiano Giuffrida, Călin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proceedings of the 27th USENIX Conference on Large Installation System Administration*, LISA'13, pages 89–104, USA, 2013. USENIX Association.
- 18 Cristiano Giuffrida, Călin Iorgulescu, and Andrew S. Tanenbaum. Mutable checkpoint-restart: Automating live update for generic server programs. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 133–144, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2663165.2663328.
- 19 Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. *SIGARCH Comput. Archit. News*, 41(1):279–292, March 2013. doi:10.1145/2490301.2451147.
- 20 Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- 21 Google. Angularjs. <https://angularjs.org/>, 2018. Accessed: 2021-04-14.


- 22 Google Inc. API Reference – Chrome Developers. <https://developer.chrome.com/docs/extensions/reference/>. Accessed: 2022-01-04.
- 23 Google Inc. JsAction repository. <https://github.com/google/jsaction>. Accessed: 2022-01-04.
- 24 Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proceedings of the ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, October 2012.
- 25 M. He, G. Wu, H. Tang, W. Chen, J. Wei, H. Zhong, and T. Huang. X-check: A novel cross-browser testing service based on record/replay. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 123–130, 2016.
- 26 Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE 2013)*, pages 612–621, May 2013.
- 27 Petr Hosek and Cristian Cadar. Varan the unbelievable: An efficient n-version execution framework. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*, pages 339–353, March 2015.
- 28 Linghua Jin. Stroop effect color game build with angularjs. <https://github.com/linghuaaj/Angular-ColorGame>, 2016. Accessed: 2021-04-14.
- 29 Aryeh Gregor Johannes Wilm. execommand – unofficial draft 13 april 2021. <https://w3c.github.io/editing/docs/execCommand/>, 2021. Accessed: 2021-04-14.
- 30 Jevgeni Kabanov and Varmo Vene. A thousand years of productivity: the jrebel story. *Software: Practice and Experience*, 44(1):105–127, 2014. doi:10.1002/spe.2158.
- 31 Brian Kirchoff. Nicedit – wysiwyg content editor, inline rich text application. <https://nicedit.com/>, 2008. Accessed: 2021-04-14.
- 32 Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016*, pages 431–442. Institute of Electrical and Electronics Engineers, Inc., September 2016. doi:10.1109/DSN.2016.46.
- 33 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. doi:10.1145/359545.359563.
- 34 Liming Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, 'Highlights from Twenty-Five Years'*, pages 113–, 1995. doi:10.1109/FTCSH.1995.532621.
- 35 Linux Foundation. ptrace – linux standard base core specification 4.1. http://refspecs.linux-foundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/baselib-ptrace-1.html, 2010. Accessed: 2021-04-14.
- 36 Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 31, USA, 2009. USENIX Association.
- 37 Matthew Maurer and David Brumley. Tachyon: Tandem execution for efficient live patch testing. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 617–630, Bellevue, WA, August 2012. USENIX Association. URL: <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/maurer>.
- 38 Meta Platforms, Inc. React – A JavaScript Library for building user interfaces. <https://reactjs.org/>. Accessed: 2022-01-04.
- 39 James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of NSDI*. USENIX, April 2010. URL: <https://www.microsoft.com/en-us/research/publication/mugshot-deterministic-capture-and-replay-for-javascript-applications/>.

- 40 Mozilla Inc. Browser Extensions – Mozilla MDN. <https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions>. Accessed: 2022-01-04.
- 41 Mozilla Inc. Firefox Public Data Report. <https://data.firefox.com/dashboard/user-activity>. Accessed: 2022-01-04.
- 42 Robert O’Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Lightweight user-space record and replay. *CoRR*, abs/1610.02144, 2016. [arXiv:1610.02144](https://arxiv.org/abs/1610.02144).
- 43 Luís Pina, Anastasios Andronidis, and Cristian Cadar. Freeda: Deploying incompatible stock dynamic analyses in production via multi-version execution. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF ’18. ACM, May 2018.
- 44 Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. MVEDSUa: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the ACM 24th Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’19. ACM, April 2019.
- 45 Luís Pina and João Cachopo. Atomic dynamic upgrades using software transactional memory. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, Hot-SWUp. IEEE, June 2012.
- 46 Luís Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. A dsl approach to reconcile equivalent divergent program executions. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC ’17. USENIX, July 2017.
- 47 Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSU for java on a stock JVM. In *Proceedings of the ACM 2014 International Conference on Object-Oriented Programming Languages, Systems, and Applications*, OOPSLA ’14. ACM, October 2014.
- 48 Weizhong Qiang, Feng Chen, Laurence T. Yang, and Hai Jin. Muc: Updating cloud applications dynamically via multi-version execution. *Future Generation Computer Systems*, 74:254–264, 2017. [doi:10.1016/j.future.2015.12.003](https://doi.org/10.1016/j.future.2015.12.003).
- 49 Rafael Robayna. Canvas painter. <http://caimansys.com/painter/>, 2006. Accessed: 2021-04-14.
- 50 Ugnius Rumsevicius, Siddhant Venkateshwaran, Ellen Kidane, and Luís Pina. Artifact for SINATRA: Stateful Instantaneous Updates for Commercial Browsers through Multi-Version eXecution, February 2023. [doi:10.5281/zenodo.7647070](https://doi.org/10.5281/zenodo.7647070).
- 51 Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs, editors, *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, pages 33–46. ACM, 2009. [doi:10.1145/1519065.1519071](https://doi.org/10.1145/1519065.1519071).
- 52 Jacob Seidelin. DOMTRIS – A DHTML Tetris clone. <https://web.archive.org/web/20140805202021/http://www.nihilogic.dk/labs/tetris/>, 2014. Accessed: 2021-04-14.
- 53 StatCounter GlobalStats. Desktop Browser Version Market Share Worldwide. <https://gs.statcounter.com/browser-version-market-share/desktop/worldwide/#daily-20201001-20201201>. Accessed: 2022-01-04.
- 54 J. R. Stroop. Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, 1935. [doi:10.1037/h0054651](https://doi.org/10.1037/h0054651).
- 55 Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A VM-centric approach. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 2009.
- 56 Al Sweigart. Pyautogui documentation. <https://pyautogui.readthedocs.io/en/latest/>, 2019. Accessed: 2021-04-14.
- 57 John Vilck, James Mickens, and Mark Marron. A gray box approach for high-fidelity, high-speed time-travel debugging. Technical Report MSR-TR-2016-7, Microsoft, June 2016. URL: <https://www.microsoft.com/en-us/research/publication/gray-box-approach-high-fidelity-high-speed-time-travel-debugging/>.

- 58 Stijn Volckaert, Bart Coppens, Bjorn De Sutter, Koen De Bosschere, Per Larsen, and Michael Franz. Taming parallelism in a multi-variant execution environment. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17*, pages 270–285, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064178.
- 59 Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 167–179, Denver, CO, June 2016. USENIX Association. URL: <https://www.usenix.org/conference/atc16/technical-sessions/presentation/volckaert>.
- 60 W3C. DOM – Living Standard – Section 4.3: Mutation Observers. <https://dom.spec.whatwg.org/#mutation-observers>. Accessed: 2022-01-04.
- 61 W3C. HTML – Living Standard – Section 10: Web workers. <https://html.spec.whatwg.org/multipage/workers.html#workers>. Accessed: 2022-01-04.
- 62 W3C. HTML – Living Standard – Section 12: Web storage. <https://html.spec.whatwg.org/multipage/webstorage.html#webstorage>. Accessed: 2022-01-04.
- 63 W3C. WebRTC 1.0: Real-Time Communication Between Browsers. <https://w3c.github.io/webrtc-pc/>. Accessed: 2022-01-04.
- 64 Web Hypertext Application Technology Working Group (WHATWG). Html living standard – 4.12.5 the canvas element. <https://html.spec.whatwg.org/multipage/canvas.html#the-canvas-element>, 2021. Accessed: 2021-04-14.
- 65 Thomas Würthinger, Danilo Ansaloni, Walter Binder, Christian Wimmer, and Hanspeter Mössenböck. Safe and atomic run-time code evolution for java and its application to dynamic aop. *SIGPLAN Not.*, 46(10):825–844, October 2011. doi:10.1145/2076021.2048129.
- 66 Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ '10*, pages 10–19, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1852761.1852764.
- 67 Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using replicated execution for a more secure and reliable web browser. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. URL: <https://www.ndss-symposium.org/ndss2012/using-replicated-execution-more-secure-and-reliable-web-browser>.

An Efficient Vectorized Hash Table for Batch Computations

Hesam Shahrokhi  
University of Edinburgh, UK

Amir Shaikhha  
University of Edinburgh, UK

Abstract

In recent years, the increasing demand for high-performance analytics on big data has led the research on batch hash tables. It is shown that this type of hash table can benefit from the cache locality and multi-threading more than ordinary hash tables. Moreover, the batch design for hash tables is amenable to using advanced features of modern processors such as prefetching and SIMD vectorization. While state-of-the-art research and open-source projects on batch hash tables made efforts to propose improved designs by better usage of mentioned hardware features, their approaches still do not fully exploit the existing opportunities for performance improvements. Furthermore, there is a gap for a high-level batch API of such hash tables for wider adoption of these high-performance data structures. In this paper, we present Vec-HT, a parallel, SIMD-vectorized, and prefetching-enabled hash table for fast batch processing. To allow developers to fully take advantage of its performance, we recommend a high-level batch API design. Our experimental results show the superiority and competitiveness of this approach in comparison with the alternative implementations and state-of-the-art for the data-intensive workloads of relational join processing, set operations, and sparse vector processing.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Computer systems organization → Single instruction, multiple data

Keywords and phrases Hash tables, Vectorization, Parallelization, Prefetching

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.27

Acknowledgements The authors would like to thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh.

1 Introduction

Hash tables are one of the most important data structures in programming. They are widely used in high-performance analytics workloads including database query processing, sparse linear algebra, graph processing, and computer networks. Besides the great efforts in algorithmic improvement of hash tables [10, 18, 20], the recent advances in modern processors, further motivated the research on high-performance hash tables that leverage the hardware characteristics including parallelization, prefetching, and vectorization.

Previous research [19] has shown that batch operations (e.g., batch lookups) on a hash table result in higher performance in comparison with ordinary scalar-parameter operations. This is because of the improved cache locality and the freedom given to the hash table designer for hand-tuning the code, which is not available when dealing with ordinary scalar-parameter operations over hash tables. Thus, hash tables with batch operations have gained more attention; both research [12, 14, 19, 23, 22, 32] and open-source projects [1, 3, 6] have proposed hash table designs with the support for batch lookups, insertions, and deletions.



© Hesam Shahrokhi and Amir Shaikhha;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 27; pp. 27:1–27:27

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Table 1** A summary of state-of-the-art batch lookups in hash tables. H: Horizontal Vectorization.

Approach	Parallelization	Prefetching	SIMD Vectorization
Hirota [3], R hashmap [6]	○	○	○
DPDK [1]	●	●	● (H)
Cuckoo++ [23]	○	●	● (H)
Polychroniou et al. [19]	●	○	●
<i>Vec-HT (this paper)</i>	●	●	●

The current literature on batch hash tables considers the following hardware features:

- Parallelization: The batch of inputs is divided into separate partitions that are processed in parallel [1, 19].
- Prefetching: The memory prefetching feature of processors is used to hide the latency of frequent memory accesses for a group of elements [1, 2, 9, 23].
- Vectorization: The Single Instruction, Multiple Data (SIMD) instructions of the processing unit are used for faster processing of a vector of inputs [1, 2, 9, 19, 23].

Although the existing approaches made noticeable efforts on improving the performance of batch hash tables, none of them fully exploits all of the three optimization dimensions mentioned above (cf. Section 2). The existing approaches for vectorization can be categorized into two classes (cf. Figure 1): (1) *Horizontal Vectorization*, where the SIMD instructions are used for the operations on a single input over multiple hash table entries [1, 9, 23], and (2) *Vertical Vectorization*, where the SIMD instructions are used for the operations on an input batch over single hash table entries [19]. The first approach is not inherently batch based; it can be applied to ordinary hash tables. By conducting intensive benchmarks, Polychroniou et al. [19] and Shankar et al. [30] have shown that vertical vectorization is faster than the horizontal approach. However, prefetching has only been considered for horizontal vectorization [1, 23], and the only existing implementation of the vertically vectorized approach [19] does not support prefetching.

This paper makes the following contributions:

- We present Vec-HT, the first batch hash table that is fully optimized in all three dimensions; Vec-HT is a multi-threaded, vertically-vectorized, and prefetching-enabled hash table that can be used for high-performance data analytics. We explain the architecture and the high-level batch API of Vec-HT in Section 3.
- Previous research has shown that in most high-performance use cases, optimizing the lookup performance is more important than the other operations (Section 2). Thus, the focus of this research is on batch lookups. We show the design decisions, the applicable optimizations, and the way we combine them for efficient batch lookups in Section 4.
- We consider several data analytics tasks that can benefit from batch hash tables, such as hash-join in query processing, set processing, and sparse vector operations (Section 5).
- We present the implementation challenges we faced (e.g., memory management and parallel iteration) and how we addressed them in Section 6.
- We experimentally evaluate (Section 7) the performance of our hash table on a set of micro benchmarks across different use case domains. Our results show that Vec-HT outperforms the state-of-the-art batch and non-batch hash tables.

2 Background and Related Work

In this section, we introduce the main concepts and techniques for building high-performance hash tables while summarizing the previous efforts in this area of research. Table 1 presents a summary of the state-of-the-art in batch hash tables.¹ To position the contributions of this paper, our approach is also appended to the table.

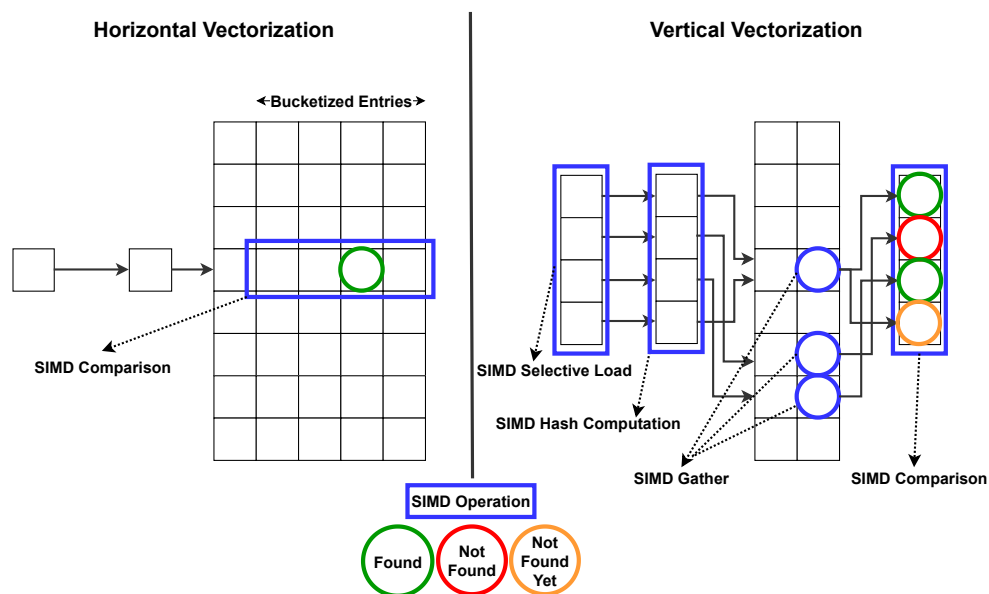
Batch Hash Tables. Batch Hash Tables accept a vector of keys or key/value pairs as their API arguments and do the operation on the inputs in batch. By taking batch inputs, the batch hash tables benefit from the cache locality and are also amenable to the use of Single Instruction, Multiple Data (SIMD), parallelism, and prefetching. The SIMD is a hardware feature that allows the simultaneous execution of an operation on a vector of values. On the other hand, prefetching is a hardware feature that allows the program to request future memory accesses in advance and asynchronous to the other computations. We will cover the more-detailed definitions of these two concepts later in this section.

Among the open-source projects [1, 3, 6], Hirola [3] presents a fast batch hash table written in C which is an alternative for `dict/set` in Python. Similarly, by considering the fact that most values in the R language are vectors and matrices, R-hashmap[6] presents a batch hash table for R, which is built over some existing ordinary hash tables in C. These two libraries are only using the cache-locality of the batch input as a way of performance improvement. As a more advanced open-source project, DPDK [1] offers a specific-purpose batch hash table for networking use cases. It is both SIMD-aware and prefetch-enabled.

There is also state-of-the-art research [9, 12, 14, 19, 22, 23, 32] on batch hash tables. Some of the approaches [12, 14, 32] are only using batch processing to benefit from the cache locality and prefetching while others [9, 19, 22, 23] use SIMD techniques on a vector of inputs. Similar to DPDK [1], Horton [9] and Cuckoo++[23] have focused on improving the performance of batch hash tables by applying SIMD and prefetching techniques to a specific type of SIMD-aware batch hash table designs called Bucketized Cuckoo Hash Tables (BCHTs). On the other hand, Polychroniou et al. [19] present a generic design for SIMD-aware batch hash tables and compare the performance of different design decisions by doing intensive benchmarks. To answer some of the open questions in vectorized hash tables, [30] conducted a survey on state-of-the-art and conducted micro-benchmarks to position each work with respect to the others.

SIMD-Aware Batch Hash Tables. To use SIMD features of a CPU in an operation (logical, arithmetic, memory, etc.), we first need to construct a vector of operands that fit the CPU register size. Then, the prepared register could concurrently process the data by using SIMD instructions. Modern CPUs offer even more advanced SIMD instructions such as selective load/store and scatter/gathers. Selective load/store makes the parallel optional load/store from/to a contiguous memory location possible by accepting a mask register. The gather/scatter operations provide the ability to load/write from/into different parts of the memory in parallel. At the time of writing this paper, the SIMD scatter is not widely adopted, and is only provided by specific hardware (e.g., Intel Xeon Phi).

¹ Although in the literature the terms batch and vectorized are used interchangeably, for the sake of consistency, in this paper, we use the term batch to refer to a collection of elements, and vectorized to refer to SIMD vectors.



■ **Figure 1** The comparison between horizontal and vertical vectorization.

As mentioned earlier, SIMD techniques can be used to improve the performance of batch hash tables. In high-level, the usage of SIMD in hash tables could be categorized in two: (1) Horizontal Vectorization and (2) Vertical Vectorization. A simplified visualization of these two approaches is depicted in Figure 1.

In horizontal vectorization, each cell of the hash-table entries array is bucketed into N inner cells. Then, while doing a lookup on the hash table and after computing the hash value, using the SIMD logical operations, the lookup algorithm can concurrently check the value of N bucket keys. This is much faster than having only one key per cell. By using this approach the hash table load factor can be improved without increasing the average lookup time. Regardless of the mentioned benefits of horizontal vectorization, it is wasteful if we expect to look up fewer than N buckets on average per probing key [19]. One of the famous open-addressing collision resolution algorithms in hash tables is Cuckoo Hashing [18]. A BCHT, as defined earlier, is actually the horizontally-vectorized version of cuckoo hashing. Many of the existing approaches on SIMD-aware batch hash tables [9, 22, 23] are in fact the improved version of BCHTs. Besides presenting a BCHT approach, Polychroniou et al. [19] also proposed and compared other horizontally-vectorized hash tables based on double hashing and linear probing hashing schemes.

Vertical vectorization [19] is a more generalizable but more complex approach to benefit from SIMD in batch hash tables. It is more generalizable because it does not change the inner structure of a hash table. However, it is more complex as it needs the collision-resolution algorithm to be translated into SIMD operations. Contrary to horizontal vectorization, in this approach, the input of a lookup operation must be a vector of probing keys. In each vectorized lookup, the vertical approach will pass a vector (of register size) of inputs through the lookup process and by using mask registers and advanced SIMD features (like SIMD permutations) probe those keys at the same time. When the status of a key lookup is determined (found/not-found), its related CPU-register lane will be assigned to the next key

in the batch of keys that are waiting to be processed. By conducting different experiments, Polychroniou et al. [19] and Shankar et al. [30] have shown that vertical vectorization yields higher performance than the horizontal approach. In vertical vectorization, since the hashing scheme must be translated into SIMD code, we need to use gathers and scatters to read/write from/to different entries of a hash table. As mentioned earlier, the scatter instruction is only available in limited types of processors hence the vertically-vectorized insertions can only be implemented on specific hardware.

The third category for SIMD-aware batch hash table design is a hybrid approach. However, the experiments in [30] show that the results of mixing the vectorized and horizontal vectorization approaches will not further improve the performance.

Besides the SIMD-aware vectorization methods discussed above, SIMD operations can also be used in the development of hash functions in any hash table [2]. Although this approach can improve the performance of hashing, it is orthogonal to the scope of this paper.

Prefetching-Enabled Hash Tables. Modern CPUs support hardware and software prefetching. Prefetching improves the performance of a program by amortizing the costs of memory access over time (in parallel to running computations).

Hardware prefetching is automatically enabled by the compiler and executed by the CPU when long and contiguous access to memory (e.g. iteration on a large vector) is requested by the program. The developer does not have much control over the hardware prefetching. On the other hand, in software prefetching, the developers can issue on-demand asynchronous prefetching commands to prefetch their future memory accesses.

In hash tables, regardless of the hashing scheme, accessing entries is based on the value of the computed hash for each provided key. This is an example of random access to a non-contiguous memory that can be improved by using the software prefetching. To have an effective prefetching in hash tables we need (1) a batch of operations and (2) a large hash table. The batch of operations provides enough computational tasks for the CPU while the prefetching instructions for future memory accesses are being processed. Also, if the hash table is not large enough, its content can entirely fit into the CPU cache. This cancels the benefits of prefetching and only puts its overheads on the CPU.

The effects of using software prefetching in hash tables have been studied in [11, 23, 32]. In the networking community, Scouarnec et al. [23] and Zhou et al. [32] have shown the effects of using prefetching on network-specific batch hash tables. They proposed different approaches and improvements in applying prefetching on BCHTs. In the database community, by proposing two generic techniques of using prefetching, Chen et al. [11] have shown their impact on the performance of relational hash joins. Among the off-the-shelf open-source hash tables, we found phmap [5] as the one that provides a `prefetch_hash` API in its interface. However, it delegates the responsibility of using this API (and designing a good prefetching strategy) to the developer who is not necessarily a system-level developer. There exist challenges in combining software prefetching with vectorization in the context of hash tables. We cover these challenges and their related design decisions in Section 4.3.

Parallel (Concurrent) Hash Tables. There is a long tail of research and open-source projects on parallel (i.e., multi-threaded) hash tables. The state-of-the-art systems [5, 7, 13, 15] have tried to enable concurrent insertion, lookup, and deletion on hash tables. These approaches can generally be divided into two categories: (1) the approaches that resolve the contentions using lock-based mechanisms, and (2) the lock-free hash tables that use atomic instructions, such as Compare-and-Swap (CAS) as their synchronization mechanism. Although these

parallel hash tables offer better performance in comparison with the sequential hash tables, they are not fully exploiting the advanced features of modern hardware such as SIMD awareness and prefetching. This is due to the lack of a batch API.

Although most of the batch hash tables offer batch insertions or deletions, previous research has shown that in most of the high-performance use cases (e.g. join processing in relational algebra, vector/tensor processing in linear algebra, and packet processing in computer networks), the amortized cost of insertions is negligible in comparison with the overall cost of highly frequent (or even endless and continuous) lookups [9]. Thus, in this paper, we only consider optimizing the lookup performance.

3 Architecture

In this section, we discuss the structure of Vec-HT and its high-level API.

3.1 Hash-Table Structure

The hash table consists of an array of `bucket` objects each of which contains a key (32 bits) and its related value (32 bits). As the hashing scheme, we use open addressing with linear probing (similar to [19]). We also use multiplicative hashing as our hash function.

Generally and without considering any optimization, to look up a key in the hash table, we first compute the hash of the key and then find its corresponding bucket in the array. If the key of that bucket is empty (the value of the empty key is defined during hash table initialization) we return the empty key which means “not found”. Otherwise, we check if the key in the bucket equals the probing key or not. If it is, we return the value, otherwise, we continue checking the next buckets to find an equal key or an empty bucket.²

3.2 High-Level API

To make a batch hash table more accessible to developers, we expose an easy-to-understand API. The Vec-HT namespace consists of three classes. A batch hash table class (`lp_map`) that is currently designed by having the open-addressing linear-probing hashing scheme (Figure 2), a batch-iterator class (`iter_batch`) that defines the data type containing the result of batch lookup, which also supports parallel iterations (Figure 3), and the `bucket` class (cf. Section 3.1) that is related to each entry of the hash table.

Batch Hash Table Class. The constructor of `lp_map` takes three arguments. The first one (`size`) is for setting the maximum number of elements that will be inserted into the hash table. The second parameter sets the group size for the internal prefetching. And the third parameter (`threads`), determines the number of threads (cores when the Hyper-Threading is disabled) to enable concurrent batch processing.

The methods exposed by the API of `lp_map` are categorized into two sets: non-batch and batch methods. The non-batch methods include `insert`, `find` that are similar to the standard hash table interfaces such as `std::unordered_map`. These methods give the developers the freedom of using Vec-HT without batch processing.

There are five batch-based methods. The method `insert_batch` inserts an array of keys and their related values into the hash table. The remaining methods are related to vectorized lookup which is the main focus of this work (cf. Section 2). The method `find_batch` accepts

² Currently, due to the restrictions imposed by SIMD-vectorization, Vec-HT only supports 32-bit integer keys (similar to [1, 19, 23]).

```

namespace vec_ht
{
    using K = uint32_t;
    using V = uint32_t;
    using P = uint32_t;
    // ---- Linear-Probing Batch Hash Table Class ----
    class lp_map
    {
        // ...
    public:
        lp_map (size_t size, size_t group_size=64, size_t threads=1);
        // ---- Non-Batch APIs ----
        inline bool insert (const K& key, const V& value);
        inline bucket* find (const K& key);
        // ---- Batch APIs ----
        inline size_t insert_batch (uint32_t* keys, uint32_t* values,
            size_t size);

        inline size_t find_batch (uint32_t* keys, size_t size,
            bool complement, iter_batch* res_it);

        inline size_t find_batch_apply (uint32_t* keys, size_t size,
            bool complement,
            std::function<void(K& key, V& value)>const& f);

        inline size_t zip (uint32_t* keys, uint32_t* payloads,
            size_t size, bool complement, iter_batch* res_it);

        inline size_t zip_apply (uint32_t* keys, uint32_t* payloads,
            size_t size, bool complement,
            std::function<void(K& key, V& value, P& payload)>const& f);
    };
}

```

■ **Figure 2** High-level API of Vec-HT in C++.

three arguments: (1) an array of keys to look up, (2) the size of that array, (3) a boolean flag called `complement` that is used to request for the not-found elements instead of the successfully-found ones, and (4) an object of a Vec-HT-specific class called `iter_batch`. The `iter_batch` class is responsible for keeping the results of a vectorized lookup and making the (parallel) iterations over them possible. The other method in `lp_map` is `zip`. Although it is not a usual API for ordinary hash tables, we found it very useful in the case of batch hash tables. This method, similar to the `find_batch`, does a lookup for the provided array of keys. However, it takes one additional argument; `payloads` assigns one value to each key in the keys array. When the method `zip` is called, the result also contains the related payloads of the found keys. The `iter_batch` class can also keep the results of a `zip` API. We show how `zip` can be used in practice in Section 5.

The remaining useful APIs are `find_batch_apply` and `zip_apply`. These APIs do the same job as their related discussed APIs. However, a user can pass their customized lambda function to be applied on the tuples of key-values (or key-value-payloads) whenever a match is found. As a result, there is no need to pass a `iter_batch` object to these APIs since it is

```

namespace vec_ht
{
    // Container and Iterator Class for find_batch/zip Results
    class iter_batch
    {
    // ...
    public:
        iter_batch (size_t max_size, size_t threads,
            bool for_zip=false)

        K** get_keys();
        V** get_values();
        P** get_payloads();

        inline void foreach
            (std::function<void(K& key, V& value)> f);

        inline void foreach
            (std::function<void(K& key, V& value, P& payload)> f);

        inline void foreach_parallel
            (std::function<void(K& key, V& value)> f);

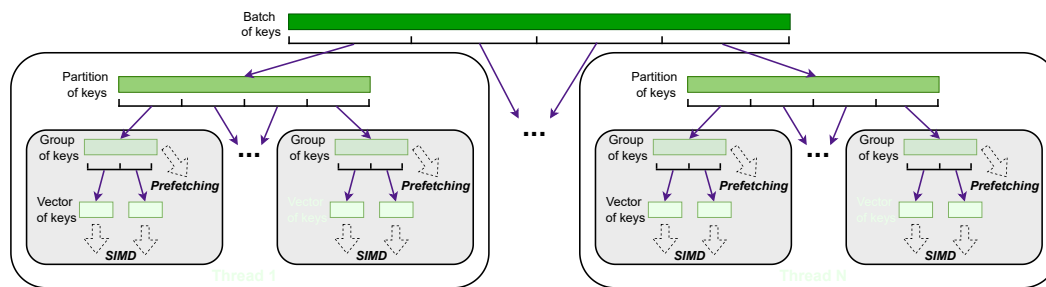
        inline void foreach_parallel
            (std::function<void(K& key, V& value, P& payload)> f);
    };
}

```

■ **Figure 3** High-level API of batched iterator in C++.

the user's responsibility to handle the output. These two APIs can improve the performance of pipelined analytical tasks because they eliminate the need for the materialization of intermediate results (`iter_batch`). In other words, using these APIs, the user can fuse the batch lookups with the following operations in the pipeline. This is especially useful in the context of pipelined analytical query processing [25, 29, 17].

Batch Iterator Class. The `iter_batch` class can be constructed by passing (1) an upper bound on the size of the results, (2) the number of threads (it must be the same as the one in `lp_map`), and (3) a boolean that shows if we want to pass this object to a `zip` or a `find_batch` API. By receiving these parameters, the memory needed for the storage of parallel-processed results will be allocated. Then, the class is ready to be sent to the methods `find_batch` or `zip` in a *destination-passing style* [26, 31]. The destination-passing style improves the performance of computational workloads by bringing the memory-allocation overheads out of the performance-critical part of the workload. The `iter_batch` class has also two overloads of `foreach`. After the execution of `find_batch` or `zip`, their relevant `foreach` method can be used for a sequential iteration over the results. The `foreach` method takes a lambda function that will be applied to each of the stored results in the `iter_batch`. Similar to the `foreach` methods, `iter_batch` also offers `foreach_parallel` methods that use multiple threads to apply the provided lambda function on the stored results. In the `foreach_parallel`



■ **Figure 4** The architecture of batch lookup in Vec-HT.

```
template<typename FUNC_TYPE>
inline size_t parallel_dispatcher(uint32_t* keys,
                                uint32_t* payloads,
                                size_t size,
                                bool complement,
                                FUNC_TYPE func,
                                iter_batch* res_iter)
```

■ **Figure 5** The signature of `parallel_dispatcher`, a function used internally for parallelization.

methods, since they are internally implemented based on `tbb::parallel_for_each` [7], the developer can also use parallel containers such as `tbb::enumerable_thread_specific` or any other off-the-shelf parallel container to handle storing/aggregation of lambda outputs.

4 Design

In this section, we discuss the design decisions behind the optimizations in our approach and show how they relate to each other. Figure 4 shows the architecture of a batch lookup in Vec-HT. As it is shown in this Figure, the batch input is partitioned into smaller chunks on different levels and for different optimization purposes. In this section, these levels of input partitioning and the rationale behind them will be covered.

4.1 Parallel Processing

To make the most out of the multi-core processor, in case of a batch lookup, we partition the input batch of keys and assign each partition to a thread for parallel batch processing. When the `find_batch` or `zip` methods are called, they call the `parallel_dispatcher` method internally. This lower-level method is responsible for managing the threads needed for the computation and passing them the contextual information. The interface of `parallel_dispatcher` is shown in Figure 5.

The `parallel_dispatcher` method takes six arguments:

1. the keys that we want to look up in the hash table,
2. the associated payloads (set to `NULL` if the caller method is `find_batch`),
3. the size of the `keys`,
4. the `complement` flag (cf. Section 3.2),
5. the lambda function that is passed when the user calls `find_bath_apply` or `zip_apply`,
6. the `iter_batch` which is passed in case of calling `find_batch` or `zip`.

4.2 SIMD-Awareness

In this section, we explain the vertical-vectorization approach for batch lookups at a high level and refer the interested reader to Polychroniou et al. [19] for more details.

Suppose that a number of W keys can be stored in a CPU register. When the batch lookup starts, W keys of the input vector will be fetched into the `keys` register. To load the input keys, we use the selective load SIMD operation. This operation uses a mask to select which lanes of the target register must be filled with the new values and which of them must be set to zero. In the beginning, we define a register (`invalid`) with all lanes activated and pass it to the selective load as the mask. This means that we plan to read W new keys from the input. Then using the SIMD operators, the hash value of all the keys in `keys` is computed simultaneously and stored in the `hash` register. In Vec-HT, as we use a simple multiplicative hash function, the computation of hash values consists of logical and arithmetic SIMD operations such as vectorized multiplication and shift.

By having the hash values, we use the SIMD `gather` operation to retrieve the needed hash table entries. The `gather` operation reads multiple memory addresses (stored in a register) at the same time. Since the value of each computed hash shows the possible offset of each key in the hash table entries, we apply `gather` on the address of the hash-table entries array and the `hash` register. As a result of executing two `gather` operations, two registers for the retrieved keys (`tab_keys`) and values (`tab_vals`) will be created.

Next, based on the linear probing algorithm, we check the equality of key in `keys` and `tab_keys`. We do this check using SIMD logical operators. This check can have three different results for each lane: (1) the key is empty which means that the key is not found in the table (2) the keys are equal which means the key is found (3) the key is not empty or is not equal to the given input key and thus it needs further probing in the next rounds. For the not-found keys, we activate their relevant lanes in `invalid` register. For the found keys, we define and activate the relevant lanes in a new register called `output`. Finally, for the ones that need further probing, we create a new register called `offset`, initialize it with 0, and increment its relevant lanes by 1.

In this phase of the algorithm, we first add `out` to `invalid` and store the result in `invalid`. We do this since we are finished with both found and not-found keys and we want to fetch the new keys instead of them in the next round of lookups. Then, by using a static permutation table, we extract the permutation masks needed to align the active lanes of `invalid` and `output` to one side of the register. These permutation masks will be used in the SIMD `permute` operation, which changes the order of lanes in the register using a provided mask. First, we use the permutation mask of `out` on `out` itself and on `keys`. Now, we are ready to save the found keys to the target memory (reserved memory in `iter_batch`) using a selective store SIMD operation that its mask is the permuted `out`. The total number of output keys will also be updated at this stage. It is notable that the original vertical vectorization [19] uses a buffer to store the results temporarily and spills them to the output whenever the buffer is full.

After the work on the found keys is finished, we count the active lanes in `invalid` to know how many new keys are needed to be fetched. Then, we apply the `invalid` permutation mask on `keys`, `hash`, and `offset` to make them ready for the next run. By starting the next round of lookups, again the new keys will be fetched based on the updated `invalid` register. It is important to mention that this time all of the hashes are re-computed and the ones with inactive lanes in `invalid` will also be incremented by `offset` to point to the next entry in the hash table.

To return the value of found keys in the table and also their related payloads we need further considerations. For the hash table values, which are stored in `tab_val`, we can permute them using the `out` permutation mask and store them in their related memory in `batch_iter`. Similarly, the payloads can be selectively loaded exactly similar to the new key. Then they will be passed through the algorithm by similar permutations, and finally will be stored in the relevant output memory.

If the size of input keys is less than W or in the case of processing the last W keys, the algorithm switches to a normal scalar (non-SIMD) lookup in the hash table and stores the result into the `batch_iter`. This is because there are not enough keys to do a safe and efficient SIMD lookup.

4.3 Prefetching and Its Adaption Challenges

Vec-HT is a prefetching-enabled vectorized approach; we apply the prefetching on top of the parallel vertical vectorization. There is a large design space for combining prefetching and SIMD vectorization. We examined this design space through micro-benchmarking (cf. Section 7). The important design parameters for this combination are as follows:

- **Standard vs Group Prefetching:** putting the prefetching commands at the beginning of the main loop of the vertical vectorization is the standard solution for adding prefetching. We compare it with another approach (Group Prefetching) proposed by Chen et al. [11] in the context of databases.
- **Group Size for Group Prefetching:** Considering the group prefetching approach, the selection of the different group sizes might affect the performance of the system.
- **Optimistic vs Pessimistic Linear Probing:** Given that we use linear probing, there are two choices for prefetching for each key. Optimistic: we consider that the probe hits the correct location on the first try (or finds the location to be empty) and does not need to probe further; thus we only prefetch the hashed location. Pessimistic: we consider the case of not having a hit and thus prefetching the next location(s) as well.
- **Memoization of Computed Hashes:** We need the hash values in two places: (1) prefetching stage, and (2) vertical vectorization stage. We have the option of memoizing the hash value in the first stage and reusing/recomputing it in the second stage.
- **Buffering:** In vertical vectorization, we can write the output into an output buffer and if it is not carefully adapted to the prefetching design, it might result in performance overheads.

Figure 6 depicts a generic and high-level algorithm for combining prefetching with vertical vectorization (based on the assumption that we take the group-prefetching approach instead of standard prefetching, which is a take-away message of micro-benchmarks in Section 7). In this algorithm, by setting the `GROUP_SIZE` parameter, we can enable the grouping loop that partitions the input keys into parts of size `GROUP_SIZE` and then run the algorithm on these smaller batches. By having a group of keys as input, before starting the vertical vectorization, we define a loop over the group keys (prefetching loop). In each iteration of the prefetching loop, we first compute the hash of W elements using the SIMD approach mentioned before. By setting the `HASH_MEMOIZE` parameter to `true`, we can store the hash values and reuse them inside the vertical vectorization algorithm. After making a decision on memoization, we raise W software prefetch commands for the address of target entries in the hash table. Here we can do the prefetching also for the next bucket by setting the `OPTIMISTIC` parameter to `false`. After finishing the prefetching loop, all the related entries are prefetched. At this stage, the vertical vectorization algorithm will be executed for the current group and if the `OUT_BUFFER` parameter is enabled, the output buffering happens.

```

foreach group in array by GROUP_SIZE {
  // prefetching stage
  foreach vector in group {
    vector_h <- simd_hash(vector)
    if(HASH_MEMOIZE)
      mem_h += vector_h
    foreach h in vector_h {
      prefetch(buckets[h])
      if(!OPTIMISTIC)
        prefetch(buckets[h+1])
    }
  }
  // vertical vectorization stage using linear probing
  while(vec_elems not probed in group) {
    if(HASH_MEMOIZE)
      vector_h <- mem_h[vec_elems]
    else
      vector_h <- simd_hash(vec_elems)
    res <- vertical_vectorization(vec_elems, vector_h)
    if(OUT_BUFFER) {
      buffer += res
      if(buffer is full)
        flush(buffer)
    }
  }
  if(OUT_BUFFER) {
    flush(buffer)
  }
}

```

■ **Figure 6** A generic algorithm showing the design space of combining prefetching with vertical vectorization.

5 Use Cases

In this section, we show the usability of our proposed batch table, by showing several high-performance data analytics use cases.

5.1 Relational Hash Join

First, the code for a join on two relations (S and R) is shown in Figure 7. We assume that the relations are stored in columnar format (i.e., struct of arrays) which is a popular design decision in high-performance query engines [17]. The code is executed using a prefetching group size of 64 on 4 threads. We keep these settings for all of the use cases covered in Section 5.

Build Phase. In the beginning, the batch hash table is initialized with the table size, group size, and the number of threads. The size is set to twice the number of the elements in the relation on the build side of the hash join (S). We do so to keep the fill ratio of the hash table less than or equal to 50%. Then, using the `insert_batch` method, all the key/value pairs from S are inserted into the hash table in a batch style.


```

// build phase
auto ht = vec_ht::lp_map(2*S_size, 64, 4);
ht.insert_batch(S_A, S_B, S_size);
// probe phase
auto res_it = vec_ht::iter_batch(R_Size/3, 4, true);
ht.zip(R_A, R_F, R_size, false, res_it);
// printing the output
res_it.foreach_parallel(
[] (auto& key, auto& value, auto& payload){
    std::cout << "S_A/R_A: " << key << " | ";
    std::cout << "S_B: " << value << " | ";
    std::cout << "R_F: " << payload << std::endl;
});

```

■ **Figure 7** Implementation of a hash join operator (on S and R relations) using Vec-HT.

```

auto ht = vec_ht::lp_map(2*S2_size, 64, 4);
for (int i=0; i<S2_size; i++) ht.insert(S2[i], 1);
auto res_it = vec_ht::iter_batch(S1_Size/5, 4, false);
ht.find_batch(S1, S1_size, true, res_it);
res_it.foreach_parallel([] (auto& key, auto& value){
    std::cout << "Item: " << key << std::endl;
});

```

■ **Figure 8** Implementation of a Set-Difference operation ($S1 \setminus S2$) using Vec-HT.

Probe Phase. Before running the batch lookups on the hash table, we first prepare the `iter_batch` for the results. This object is initialized by setting three parameters. The first one is an upper bound for the join result size. The more precise this estimation is, the less memory allocation time is spent during the batch lookups. The second parameter is the number of threads that must be equal to the one already passed to the hash table. The last parameter is a flag that shows if the `iter_batch` object will be used in a `zip` or `find_batch` API. Next, by having the `res_it` object, we can run our `zip` method to join the relations based on the `R_A` and `S_A` columns and by considering the `R_F` column as the payload.

After the `zip` execution is finished, we use the `foreach_parallel` method of `res_it` to iterate over the join results and print them. The desired functionality (printing) is passed to the `foreach_parallel` using a user-defined lambda function.

5.2 Set Operations

As our second use case, we show the implementation of a set difference operation ($S1 \setminus S2$) using our approach in Figure 8. The sets `S1` and `S2` are stored in two arrays. The code is almost the same as the one in the relational-join example. Its main difference is in using `find_batch` instead of `zip`. It is because there is no payload to be passed into the `zip` API. Furthermore, in this example, we see the usage of `complement` parameter as we need the elements of `S1` that are not found in `S2`. To implement the set intersection we need to use the `zip` method, which is similar to the vector inner product, that is presented next.

```

auto ht = vec_ht::lp_map(2*V1_size, 64, 4);
ht.insert_batch(V1_idx, V1_val, V1_size);
auto res_it = vec_ht::iter_batch(V1_size, 4, true);
ht.zip(V2_idx, V2_val, V2_size, false, res_it);
uint32_t sum = 0;
res_it.foreach([](auto& key, auto& value, auto& payload){
    sum += value * payload;
});

```

■ **Figure 9** Implementation of an Inner-Product operation ($V1 \cdot V2$) using Vec-HT.

5.3 Sparse Vector Operations

The last use case that we cover in this section is the inner product of two vectors ($V1 \cdot V2$). The related code is shown in Figure 9. In this implementation, we again use the `zip` method, since there are payloads on the `V2` side (values of `V2` for each index). After the `zip` execution is finished, this time we do a non-parallel iteration (`foreach` API) over `res_it` to prevent contentions on the `sum` shared memory. As mentioned in Section 3.2, a developer can easily use the concurrent containers (e.g. `tbb::enumerable_thread_specific<uint32_t>`) instead of `sum` here. However, in this case, we are interested in exhibiting the usage of non-parallel `foreach`.

6 Implementation

In this section, we give a more detailed explanation of the implementation behind Vec-HT.

Attributes of `lp_map`. The attributes of `lp_map` class are shown in Figure 10. All these attributes are initialized in the class constructor. The `size`, `threads_`, and `group_size_` attributes are set to the values that are passed by the constructor. The `hash_factor_` is set to a randomly generated number. The `empty_key_` attribute is set to the maximum possible value for `uint32_t` type. Lastly, we use an array of `bucket` structs (`entries_`) as the entries of our hash table. It has been shown that using an array of structs instead of a struct of arrays does not affect the performance of hash tables [21]. The memory of this array is allocated after the calculation of `size_` attribute.

Attributes of `iter_batch`. The attributes of the `iter_batch` class are shown in Figure 11. Here `max_size_` is passed by the constructor and shows an upper bound on the number of results for a `find_batch` or `zip` method call. The `threads_` and `for_zip` attributes are given by the constructor. The next three attributes are the storage for results of a `find_batch` or `zip` method call. In the constructor, we allocate arrays of arrays to these pointers; this will allocate memory of size `max_size_` for each thread. In the case of a `find_batch` method call, we do not allocate and use the `payloads_` pointer. As the last attribute, we have `threads_res_size_` that will be extended to the size of `threads_`. Each element of this vector is used by a thread to store the size of the results for that thread. By using this attribute, the iterations over the results will be more efficient (cf. Figure 12).

Implementation of `foreach_parallel`. In Figure 12, the implementation details of `foreach_parallel` are presented. In this method, we create a range of integers from 0 to `thread_-1` and assign each number in the range to a thread. Then, by execution of a

```

namespace vec_ht
{
    class lp_map
    {
    private:
        size_t size_;
        size_t threads_;
        uint32_t group_size_;
        uint32_t hash_factor_;
        uint32_t empty_key_;
        bucket* entries_;

        template<typename FUNC_TYPE>
        inline size_t parallel_dispatcher(uint32_t* keys,
            uint32_t* payloads, size_t size, bool complement,
            FUNC_TYPE func, iter_batch* res_iter)

        template<typename FUNC_TYPE>
        inline size_t find_batch_inner(uint32_t* keys,
            uint32_t* payloads, size_t size, bool complement,
            FUNC_TYPE func, iter_batch* res_iter, size_t thread_id)
    public:
        // ...
    };
}

```

■ **Figure 10** The internal of the `lp_map` class.

`tbb::parallel_for_each` and passing the prepared range to it, we run a lambda function on each thread with `thread_id` as its single argument. In the lambda function, using the `thread_id` argument, the `max_size_` attribute of `batch_iter` class, and the vector of result sizes for each thread (`threads_res_size_`), we compute the boundaries of the result vectors that are assigned to the current thread. By having those boundaries, we can finally apply the developer-provided lambda (`func`) on each triple of `key`, `value`, and `payload` in the results assigned to this thread.

Implementation of `zip`. The implementation of the `zip` method is presented in Figure 13. To bypass the overheads of dispatching in the parallel scenario, this method (and other performance-critical methods such as `find_batch`), checks the `threads_` attribute of the current `vec_ht`. If it detects a sequential setting, then calls the internal method that is responsible for the vertical vectorization and prefetching (`find_batch_inner`). Otherwise, the method calls the `parallel_dispatcher` (cf. Section 4) to partition and dispatch the work among the pre-determined number of threads. To call either of these two internal methods, the `zip` method provides them with the appropriate arguments or `null` types where required.

Implementation of `find_batch_inner`. Figure 14 shows a simplified implementation for `find_batch_inner`. This method is the most complex method in `Vec-HT`. It operates over a subset of batch input (the partition that is assigned to each thread) and is responsible to do the following tasks:

27:16 An Efficient Vectorized Hash Table for Batch Computations

```
namespace vec_ht
{
    class iter_batch
    {
    private:
        size_t max_size_;
        size_t threads_;
        bool for_zip_;

        uint32_t** keys_;
        uint32_t** values_;
        uint32_t** payloads_;

        std::vector<size_t> threads_res_size_;
    public:
        // ...
    };
}
```

■ **Figure 11** The internal of the `iter_batch` class.

```
inline void foreach_parallel
(std::function<void(K& key, V& value, P& payload)> f)
{
    auto range = std::vector<size_t>(threads_);
    for (size_t i=0; i<threads_; i++) range[i] = i;
    tbb::parallel_for_each(range, [&](size_t thread_id)
    {
        for (size_t j=0; j<threads_res_size_[thread_id]; j++)
            func(keys_[thread_id][j],
                values_[thread_id][j],
                payloads_[thread_id][j]);
    });
}
```

■ **Figure 12** The implementation of `foreach_parallel` in `iter_batch`.

- To partition the input into group-sized batches.
- To compute and memoize the hashes for each group.
- To do the group prefetching for each group.
- To run the entire vertical vectorization algorithm for each group.
- To buffer the found keys and their related values and payloads.
- To store the results into the `iter_batch` or apply `func` over them.

We present a brief overview of the above-mentioned steps in Figure 14. The sections with high similarity to the code provided by Polychroniou [19] et al. are removed for the sake of brevity. We refer the interested reader to see those parts in the referenced work. Note that in Figure 14, the `vector_size` is a global constant (8) which is a function of the selected data-type size (32 bits) and the SIMD vector size (256 bits), computed as vector size divided by data-type size.

```

inline size_t zip (uint32_t* keys, uint32_t* payloads, size_t size,
bool complement, iter_batch* res_it)
{
    if (threads_ == 1)
        return find_batch_inner<no_func_type>(keys, payloads, size,
        complement, nullptr, res_it);
    else
        return parallel_dispatcher<no_func_type>(keys, payloads,
        size, complement, nullptr, res_it);
}

```

■ **Figure 13** Implementation of zip in lp_map.

As the last topic in this section, to implement the complement behaviour in Vec-HT, we have slightly changed the original vertical-vectorization algorithm. In the case of a complement, we replace the keys with an *invalid* status with the keys with an *output* status. In other words, the found keys are considered *invalid* and the not-found keys are the *valid* ones that must be stored in the output.

7 Evaluation

In this section, we first present our experimental setup for the evaluation. Then, we show the performance of our approach in different use case scenarios and compare its performance with various competitors.

7.1 Experimental Setup

All experiments are done on a single machine equipped with 16GB of DDR4 RAM, and an Intel Core i5-10210U 1.6GHz with 4 cores and 256KB, 1MB, and 6MB of L1, L2, and L3 cache respectively. Hyper-threading is disabled for the experiments. We have used Ubuntu 20.04.3 as OS. Our C++ code is compiled with G++ 9.4.0 using the `-O3` flag. To enable SIMD operations, we use the `-march=core-avx2` flag. All of the experiments were executed with 5 warmup rounds followed by 5 timed iterations. Then, we took the average of the timed iterations.

Workloads. To run the experiments, we use three different workloads. For the micro-benchmarks and the join experiments, we use the random data generator from [19]. By focusing on the notion of inner joins in databases, it generates two random data sets as inner and outer relations. The elements of the inner data set are inserted into the hash table creating the build side of the join. The elements of the outer data set shape the probe side of the join. The data generator accepts arguments for `inner_size`, `outer_size`, and `selectivity` of the join.

The second workload is used for the set and sparse vector experiments. The set/vector generator receives `size`, `density`, and `maximum_value` as input parameters. For each set of size N , it generates $N \times \text{density}$ unique random numbers from the range of $[0, N)$ as the value of items in the set. Similar to the sets, for the vectors, it generates unique random numbers but uses them as vector indexes. Then, using the `maximum_value` parameter, it generates random integer numbers in the range of $(0, \text{maximum_value}]$ as vector values.

27:18 An Efficient Vectorized Hash Table for Batch Computations

```
inline size_t find_batch_inner (uint32_t* keys, uint32_t* payloads,
                               size_t size, bool complement,
                               FUNC_TYPE func, iter_batch* res_iter,
                               size_t tid)
{
    // Partitioning the input keys into group-sized batches
    size_t inner_batch_size = group_size_;
    for(size_t i=0; i<size; i+=group_size_)
    {
        if (size-i<group_size_)
            inner_batch_size = size%group_size_;
        // Hash memoization and Group prefetching
        uint32_t hashes[inner_batch_size];
        for (size_t j=0; j<inner_batch_size; j+=vector_size)
        {
            // Hash computation for keys using SIMD operations
            // ...
            // Hash memoization and Group prefetching
            for (size_t k=0; k<vector_size; k++)
            {
                // Storing the hash in hashes[j+k]
                // ...
                // Prefetching the computed and stored hash
                _mm_prefetch(&entries_[hashes[j+k]], _MM_HINT_T0);
            }
        }
        // Execution of vert. vect. using memoized hash values
        // considering the "complement" flag (if enabled) ...

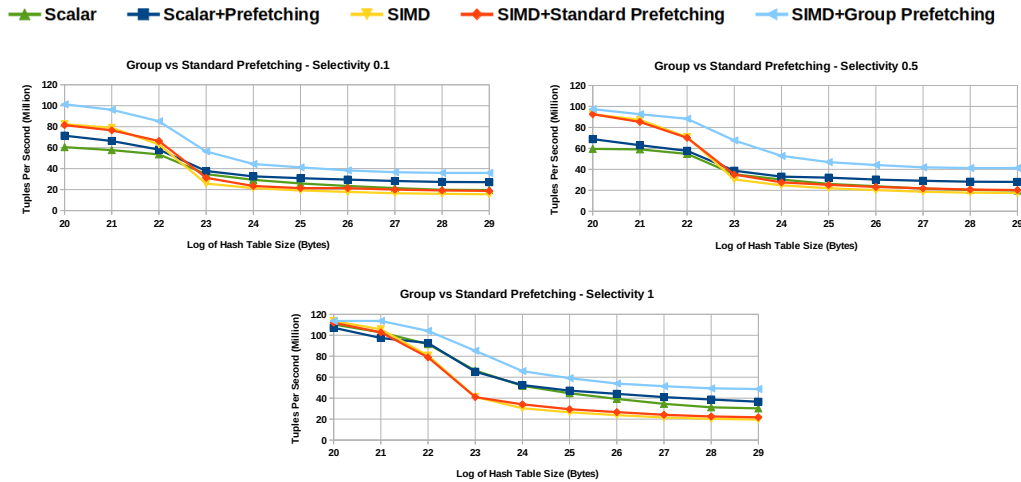
        // Buffering the matched keys, values (and payloads) ...

        // Flushing the buffer into the iter_batch container or
        // Applying the lambda function on the buffered tuples ...
    }
}
```

■ **Figure 14** A simplified representation of the `find_batch_task` implementation in `lp_map`.

As the last workload, we use the well-known TPC-H [8] benchmark with a scaling factor (SF) of 1 (1 GB of data) for the evaluation of our approach in analytical queries. It is important to note that in all of the benchmarks, we keep the fill ratio of all alternative hash tables less than or equal to 50%.

Alternatives and Competitors. In the micro-benchmarks, to evaluate our proposed approach, we compare it with (1) a scalar implementation of Vec-HT without any optimization (2) a scalar + prefetching version (3) and the vertical-vectorization approach by Polychroniou et al. [19]. For all alternatives, we consider sequential and parallel versions. As mentioned in Section 6, we reuse the code from [19] as the base for our implementations. We do not add the comparison with the approaches such as DPDK [1], as it is previously shown [19] that the BCHT approach is slower than vertical vectorization which is the basis for Vec-HT.



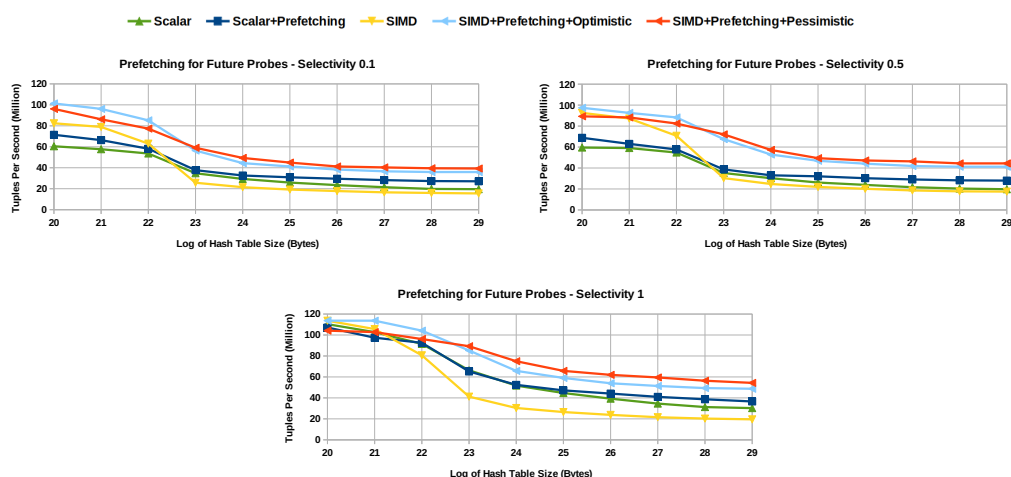
■ **Figure 15** Comparing the performance of standard vs group prefetching for combining prefetching with vertical vectorization. The charts show the number of tuples processed per second (higher is better) for a simplified relational inner-join operation with a probe size of 1,500,000 over an increasing build-side size.

We also compare our implementations with state-of-the-art hash tables. TBB [7] is a well-known parallel computation framework. We use its `tbb::concurrent_unordered_map` as one of our competitors. Libcuckoo [13] is a research project on fast parallel hash tables and we use its open-source implementation `libcuckoo::cuckoohash_map`. As the last competitors, from the open-source high-performance hash-table project phmap [5], we use its sequential and parallel data structures `phmap::flat_hash_map` and `phmap::parallel_flat_hash_map`. The implementation of Vec-HT that we use in the benchmarks has a group size of 64, taking an optimistic approach, with enabled memoization and simple buffering inside each group.

7.2 Benchmarks

In this section, we first consider the design space of combining prefetching with vectorization and show the best implementation. In addition, we evaluate the effectiveness of our implementation in comparison with scalar, scalar + prefetching, and pure vertical vectorization in a holistic micro-benchmark for hash join processing. Then, we show its superiority over the existing hash table packages in different use cases. We consider benchmarks on set and vector kernels that are largely used in big data analytics frameworks such as query processors (e.g., BigTable, SparkSQL) and linear algebra frameworks (e.g., MLlib, SystemDS, distributed TensorFlow). We finally cover benchmarks on database query processing over a selected subset of TPC-H queries, the main benchmark for analytical queries.

Standard vs Group Prefetching Micro-Benchmark. The first micro-benchmark related to the design space (cf. Section 4.3) is shown in Figure 15. In these experiments, our focus is to show the performance difference between the standard prefetching and group prefetching approaches. The results show that even though the standard way of prefetching offers performance improvements compared to non-prefetched approaches, it cannot beat the performance and scalability of the group prefetching.



■ **Figure 16** Comparing the performance of different alternatives by focusing on the optimistic and pessimistic approaches for prefetching. The workload is similar to Figure 15.

Optimistic vs Pessimistic Prefetching Micro-Benchmark. As mentioned in Section 4.3, by having a linear probing scheme in the hash table, for a given key, we can prefetch more than one bucket to improve the chance of a cache hit after an unsuccessful lookup. Although it seems to be an interesting strategy to take, the limited prefetching capability of CPUs, the variety in workload characteristics, and the parameters such as the fill ratio of the hash table can affect the benefits of this strategy. Figure 16 shows the performance results for prefetching with optimistic and pessimistic approaches. Both optimistic and pessimistic approaches perform better than the alternatives. However, for the smaller hash tables, the performance of the pessimistic approach is worse than the optimistic one and sometimes even worse than pure vertical vectorization. Thus, we decided to take the optimistic approach for our final implementation of Vec-HT.

Group-Size Micro-Benchmark. Figure 17 depicts the performance of group prefetching with different group sizes by also altering between the optimistic and pessimistic strategy. Overall, we see a performance improvement by increasing the group size, however, this improvement no longer holds after the group size of 64. The results show that selecting a group size of 64 with the optimistic strategy is a reasonable choice.

Hash Memoization Micro-Benchmark. Our last micro-benchmark investigates the effects of memoizing the hash values during the prefetching stage. Figure 18 presents the results of these experiments. In selectivities of 0.1 and 0.5, it is clear that the memorized version performs better than the non-memoized one. With a selectivity of 1, the non-memoized version does better (with a narrow improvement compared to memorized version) for larger hash tables, however, the memoized version is faster for smaller hash tables. Thus, we select the memoized version for Vec-HT.

Relational Inner Join for State-of-the-Art Hash Tables. In these experiments, using the first workload described in Section 7.1, we run a simplified relational inner-join operation using different hash table implementations to measure the performance of each approach.

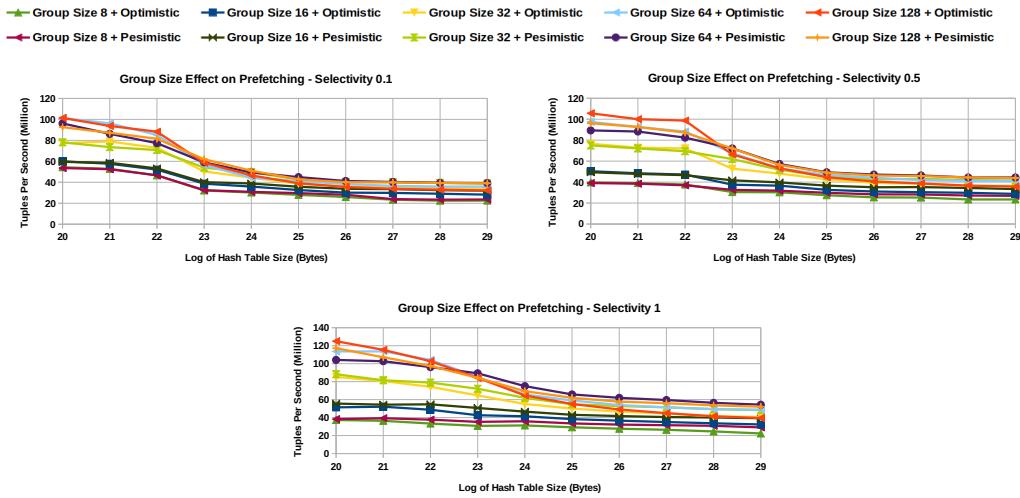


Figure 17 Comparing the performance of different combinations for group size and optimistic/pessimistic approaches for group prefetching. The workload is similar to Figure 15.

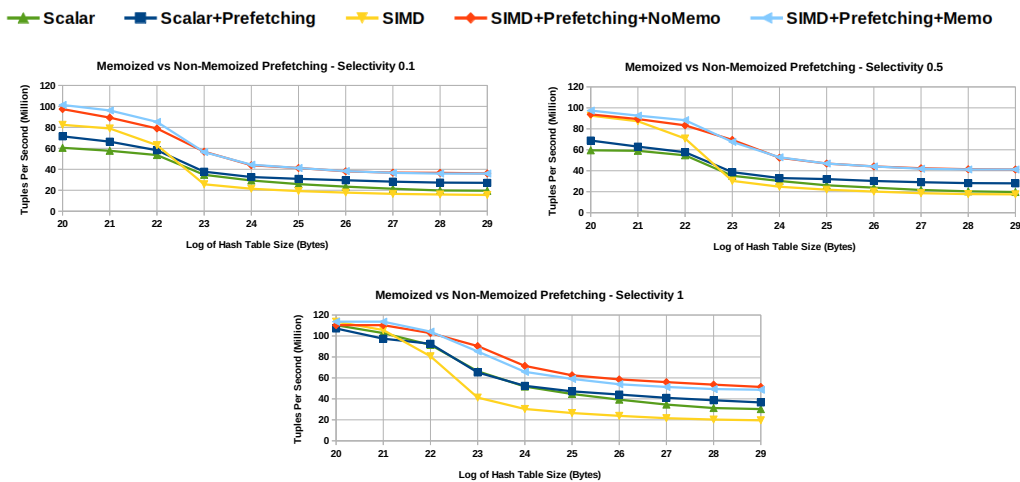
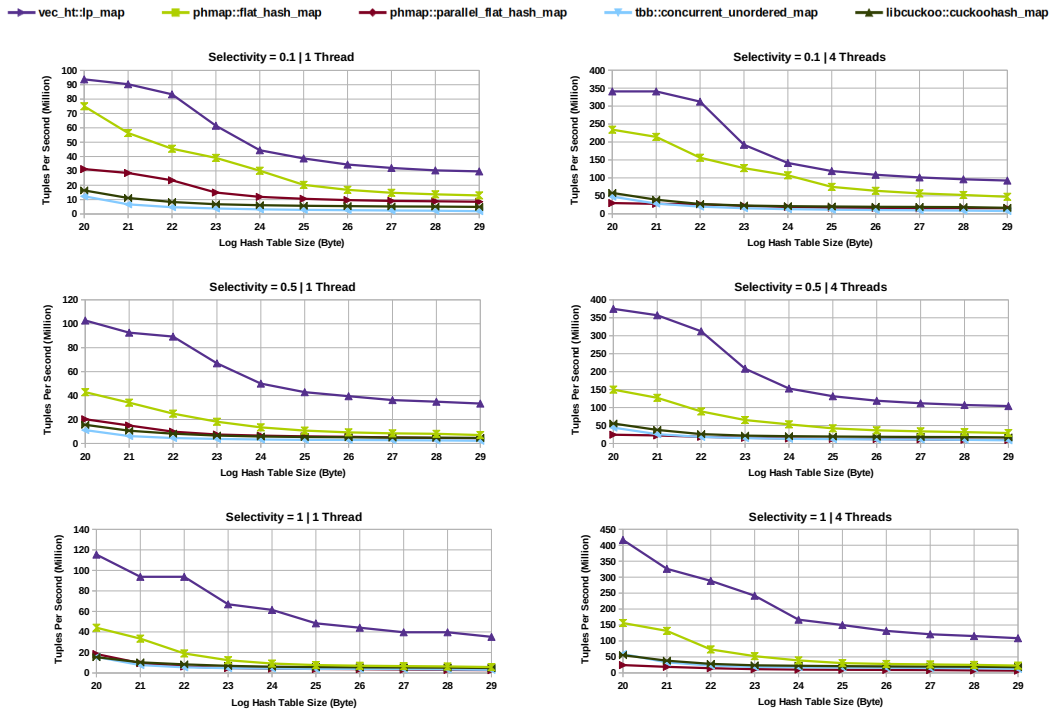


Figure 18 Comparing the performance of different alternatives by focusing on enable/disabled memoization for group prefetching. The workload is similar to Figure 15.

As it is shown in Figure 19, our approach `vec_ht::lp_map` outperforms other approaches irrespective of the build size, join selectivity or concurrency level. It is on average 5× faster on both 1- and 4-cores.

Set Operations Use Case. To show the performance of our approach in set operations, we run experiments on the set intersection and set difference. To run the operations, we iterate over the first set (S1) and look up the values in the other one (S2). Since the hash tables in our experiments (except `tbb::concurrent_unordered_map`) do not support parallel iterations, to have a more comprehensive benchmark, we keep S1 in the vector format and only make a hash table for S2. For the set-difference operation, we use an implementation similar to Section 5.2. Figure 20 depicts the results of our set experiments. In these experiments, using a fixed size for S1 and S2 and a fixed density for S1, we observe the changes in the run time of each competitor while increasing the density of S2.

27:22 An Efficient Vectorized Hash Table for Batch Computations



■ **Figure 19** The number of tuples processed per second (higher is better) for a simplified relational inner-join operation with a probe size of 1,500,000 over an increasing build-side size. Charts on each side, represent the results for the join selectivities of 0.1, 0.5, and 1 on 1 or 4 threads.

In the set-difference experiments, our approach outperforms the others with an average of $11\times$ speedup on 1 and 4 cores. Similarly, in set intersections, our approach performs better than the others excluding `phmap::flat_hash_map`. By excluding `phmap`, the proposed approach is on average $6\times$ and $5\times$ faster than the others on 1 and 4 cores, respectively. For the small S_2 sizes, when the log of S_2 density is less than or equal to -4 , the hash table can still be fitted into the L3 cache, thus the benefit of using our prefetched approach is not promising and the overall performance is near to what `phmap::flat_hash_map` offers. However, after passing that size limit, `phmap::flat_hash_map` run time goes higher while our approach keeps its good performance thanks to software prefetching.

Vector Operations Use Case. In Figure 21, the results of running experiments on vector inner-product (cf. Section 5.3) and pair-wise multiplication are shown. Similar to the set operations, here we keep V_1 in the vector format and embed V_2 into a hash table. The experiment parameters are also set to the values that we had in the set experiments. In both vector operations, our approach is still faster than the alternatives. However, since the scenario of these two vector operations is very similar to the set intersection, we see similar behaviour in the performance of our approach versus `phmap::flat_hash_map`. We explained the reason behind this behaviour in the set experiments. The experiments on set and vector operations show that our approach is a great choice when we deal with large amounts of data; while the performance of other approaches degrades with increasing the hash table size, our approach maintains good performance even for heavier workloads.

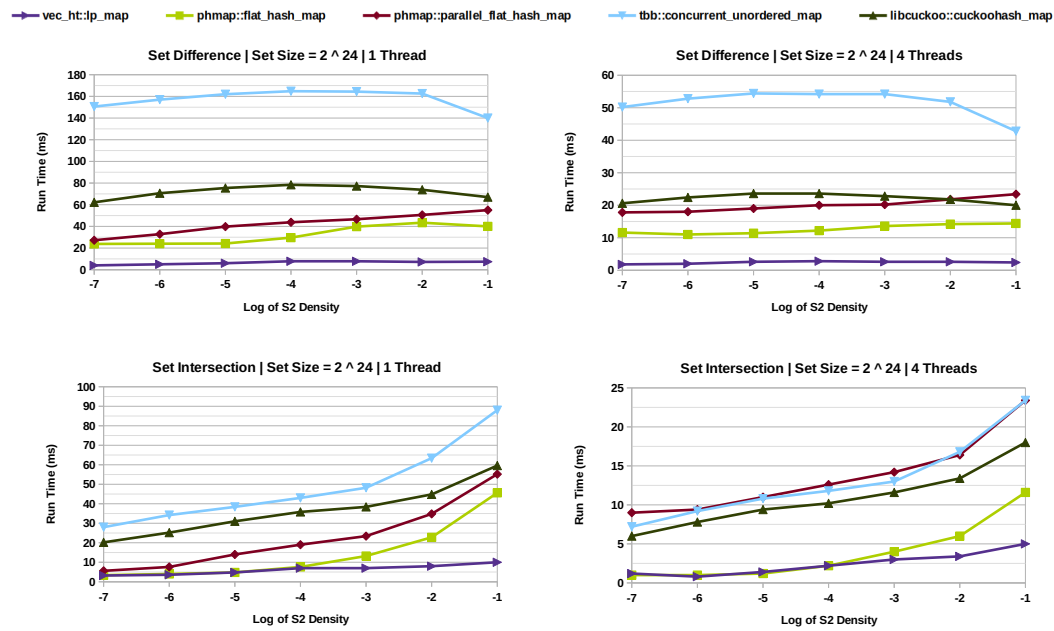


Figure 20 The run time (lower is better) for the set difference and intersection operations by varying the density of the second set on 1 and 4 threads. For both operands (S1 and S2), the size is 2^{24} . For S1, the density is set to 2^{-6} .

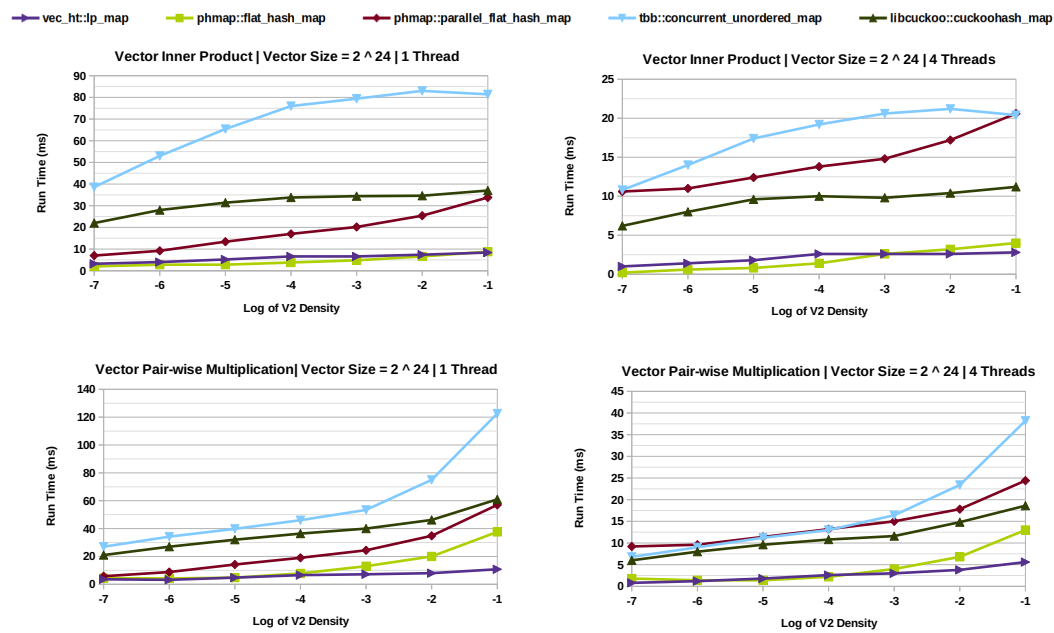


Figure 21 The run time (lower is better) for the vector inner-product and pair-wise multiplication operations by increasing the density of the second vector on 1 and 4 threads. For both operands (V1 and V2), the size is 2^{24} . For V1, the density is set to 2^{-6} .

■ **Table 2** Modified TPC-H queries we used in the experiments.

Query	SQL Code
Q4	<pre>select o_orderpriority, count(*) from orders where exists (select * from lineitem where l_orderkey=o_orderkey and l_commitdate<l_receiptdate) group by o_orderpriority</pre>
Q8	<pre>select o_year, sum(case when nation = 'BRAZIL' then volume else 0 end) / sum(volume) from (select extract(year from o_orderdate) as o_year, l_extendedprice * (1 - l_discount) as volume, n2.n_name as nation from part, supplier, lineitem, orders, customer, nation n1, nation n2, region where p_partkey = l_partkey and s_suppkey = l_suppkey and l_orderkey = o_orderkey and o_custkey = c_custkey and c_nationkey = n1.n_nationkey and n1.n_regionkey = r_regionkey and r_name = 'AMERICA' and s_nationkey = n2.n_nationkey and o_orderdate between date '1995-01-01' and date '1996-12-31') as all_nations group by o_year</pre>
Q12	<pre>select l_shipmode, sum(case when o_orderpriority = '1-URGENT' or o_orderpriority = '2-HIGH' then 1 else 0 end) , sum(case when o_orderpriority <> '1-URGENT' and o_orderpriority <> '2-HIGH' then 1 else 0 end) from orders, lineitem where o_orderkey = l_orderkey group by l_shipmode</pre>

Analytical Queries Use Case. As the last set of experiments, we use the TPC-H benchmark and dataset. The queries we consider satisfy three criteria. (1) The join-build side of the query must result in a large hash table. (2) The join-probe part of the query must be a time-consuming part of it. (3) The build-side hash table can only have integers as keys and values. Based on these criteria we selected a modified version of Q4, Q8, and Q12 (cf. Table 2). Then, we implemented a manually fine-tuned version of them in C++ by taking the query plans of HyPer [4] and using the code generator of SDQL.py [24]. We used `phmap::flat_hash_map` as the competitor for `vec_ht::lp_map`, because it is the hash table of choice behind existing query processing systems that use open source hash tables [24, 28]. We alternate between these two hash tables only in the most time-consuming join operation in the query.

Table 3 shows the results of this benchmark. Our approach performs better than its alternative almost in all these queries. However, in the 4-core setup of Q4, it results in an 11% total performance degradation. In this case, the speedup of probing is still very high (1.33×) but the lack of parallel insertions in Vec-HT resulted in a faster hash table building by the competitor, which is a promising direction for the future.

Finally, it is worth mentioning that the speedups of using Vec-HT for the original Q8 (without modification on the build side to make the hash table larger) are 0.9× and 0.85× for 1- and 4-core respectively. This means that Vec-HT is a perfect choice whenever we are facing a large volume of data resulting in the creation of a large hash table.

■ **Table 3** Performance improvements of using `vec_ht::lp_map` instead of `phmap::flat_hash_map` in the probes of the most time-consuming hash-join of TPC-H queries 4, 8, and 14.

	TPC-H Query					
	Q4		Q8		Q12	
	Total	Probe	Total	Probe	Total	Probe
1-Core Run Time <code>vec_ht</code> (ms)	246	102	354	321	260	236
1-Core Run Time <code>phmap</code> (ms)	487	261	512	465	718	533
1-Core Total Run Time Speedup	1.98×	2.56×	1.44×	1.45×	2.76×	2.25×
4-Core Run Time <code>vec_ht</code> (ms)	151	29	105	83	102	82
4-Core Run Time <code>phmap</code> (ms)	134	38	140	107	395	149
4-Core Total Run Time Speedup	0.89×	1.33×	1.34×	1.29×	3.86×	1.82×

8 Conclusion and Future Work

In this paper, we presented Vec-HT, a vectorized hash table that offers fast batch lookups backed by multi-threading, prefetching, and usage of SIMD-vectorization methods. We presented the design decisions for the structure, API, and the optimizations for high-performance batch hash table implementations. We showed the usefulness of our approach by implementing a handful of use cases using Vec-HT. Finally, by running a set of micro-benchmarks on various use case scenarios, we showed that our proposed design performs faster than the state-of-the-art approaches.

In the future, we aim to improve this approach by providing the support for complex keys and values and parallel iterations over such batch hash tables. It is also interesting to apply the current optimizations (especially vertical vectorization) to the other hashing schemes such as Cuckoo [18] and Robinhood [10] hashing. Another promising direction is to use the batch API as a wrapper for traditional hash tables and ordered dictionaries [27] to allow programmers to benefit from the batch processing offered by this API. Finally, one can integrate other SIMD query operators (e.g., selection [19, 16]) and use Vec-HT for a wider range of database analytical queries as well as sparse tensor processing.

References

- 1 Dpdk. <https://dpdk.org/>.
- 2 Highwayhash. [arXiv:1612.06257](https://arxiv.org/abs/1612.06257).
- 3 Hirola. <https://github.com/bwoodsend/hirola/>.
- 4 Hyper. <https://hyper-db.de/>.
- 5 The parallel hashmap. <https://github.com/greg7mdp/parallel-hashmap>.
- 6 R hashmap. <https://github.com/nathan-russell/hashmap>.
- 7 Threading building blocks (tbb). <https://github.com/jckarter/tbb>.
- 8 TPC-H Benchmark . <https://www.tpc.org/tpch>.
- 9 Alex D Breslow, Dong Ping Zhang, Joseph L Greathouse, Nuwan Jayasena, and Dean M Tullsen. Horton tables: Fast hash tables for {In-Memory}{Data-Intensive} computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 281–294, 2016.
- 10 Pedro Celis, Per-Ake Larson, and J Ian Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288. IEEE, 1985.
- 11 Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17–es, 2007.

- 12 Bin Fan, David G Andersen, and Michael Kaminsky. {MemC3}: Compact and concurrent {MemCache} with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.
- 13 Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys’14*, pages 1–14, 2014.
- 14 Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 1–13, 2011.
- 15 Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general (!) *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–32, 2019.
- 16 Prashanth Menon, Todd C Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11(1):1–13, 2017.
- 17 Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, 2011.
- 18 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.
- 19 Orestis Polychroniou, Arun Raghavan, and Kenneth A Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1493–1508, 2015.
- 20 Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. *Journal of the ACM (JACM)*, 59(3):1–50, 2012.
- 21 Stefan Richter, Victor Alvarez, and Jens Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.
- 22 Kenneth A Ross. Efficient hash probes on modern processors. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1297–1301. IEEE, 2007.
- 23 Nicolas Le Scouarnec. Cuckoo++ hash tables: High-performance hash tables for networking applications. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, pages 41–54, 2018.
- 24 Hesam Shaikhha and Amir Shaikhha. Building a compiled query engine in python. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction, CC 2023*, pages 180–190, 2023.
- 25 Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28:e10, 2018.
- 26 Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 12–23, 2017.
- 27 Amir Shaikhha, Mahdi Ghorbani, and Hesam Shahrokhi. Hinted dictionaries: Efficient functional ordered sets and maps. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
- 28 Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–33, 2022. doi:10.1145/3527333.
- 29 Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1907–1922, 2016.
- 30 Dipti Shankar, Xiaoyi Lu, and Dhabaleswar K DK Panda. Simdht-bench: characterizing simd-aware hash table designs on emerging cpu architectures. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 178–188. IEEE, 2019.

- 31 Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R Newton. Local: a language for programs operating on serialized data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 48–62, 2019.
- 32 Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckoo-switch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108, 2013.

Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

Amir Shaikhha ✉

University of Edinburgh, UK

Mahdi Ghorbani ✉

University of Edinburgh, UK

Hesam Shahrokhi ✉

University of Edinburgh, UK

Abstract

This paper introduces hinted dictionaries for expressing efficient ordered sets and maps functionally. As opposed to the traditional ordered dictionaries with logarithmic operations, hinted dictionaries can achieve better performance by using cursor-like objects referred to as hints. Hinted dictionaries unify the interfaces of imperative ordered dictionaries (e.g., C++ maps) and functional ones (e.g., Adams' sets). We show that such dictionaries can use sorted arrays, unbalanced trees, and balanced trees as their underlying representations. Throughout the paper, we use Scala to present the different components of hinted dictionaries. We also provide a C++ implementation to evaluate the effectiveness of hinted dictionaries. Hinted dictionaries provide superior performance for set-set operations in comparison with the standard library of C++. Also, they show a competitive performance in comparison with the SciPy library for sparse vector operations.

2012 ACM Subject Classification Software and its engineering → Functional languages; Theory of computation → Data structures design and analysis

Keywords and phrases Functional Collections, Ordered Dictionaries, Sparse Linear Algebra

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.28

Acknowledgements The authors would like to thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh.

1 Introduction

Sets and maps are two essential collection types for programming used widely in data analytics [12]. The underlying implementation for both are normally based on 1) hash tables or 2) ordered data structures. The former provides (average-case) constant-time lookup, insertion, and deletion operations, while the latter performs these operations in a logarithmic time. The trade-off between these two approaches has been heavily investigated in systems communities [7].

An important class of operations are those dealing with two collection types, such as set-set-union or the merge of two maps. One of the main advantages of hash-based implementations is a straightforward implementation for such operations with a linear computational complexity. However, naïvely using ordered dictionaries results in an implementation with a computational complexity of $O(n \log(n))$.

Motivating Example. The following C++ code computes the intersection of two sets, implemented by `std::unordered_set`, a hash-table-based set:

```
std::unordered_set<K> result;
for(auto& e : set1) {
    if(set2.count(e))
        result.emplace(e);
}
```



© Amir Shaikhha, Mahdi Ghorbani, and Hesam Shahrokhi;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 28; pp. 28:1–28:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

28:2 Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

However, the same fact is not true for ordered data structures; changing the dictionary type to `std::set`, an ordered implementation, results in a program with $O(n \log(n))$ computational complexity. This is because both the `count` (lookup) and `emplace` (insertion) methods have logarithmic computational complexity.

As a partial remedy, the standard library of C++ provides an alternative insertion method that can take linear time, if used appropriately. The `emplace_hint` method takes a hint for the position that the element will be inserted. If the hint correctly specifies the insertion point, the computational complexity will be amortized to constant time.¹

```
std::set<K> result;
auto hint = result.begin();
for(auto& e : set1) {
    if(set2.count(e))
        hint = result.emplace_hint(hint, e);
}
```

However, the above implementation still suffers from an $O(n \log(n))$ computational complexity, due to the logarithmic computational complexity of the lookup operation (`count`) of the second set. Thanks to the orderedness of the second set, one can observe that once an element is looked up, there is no longer any need to search its preceding elements at the next iterations. By leveraging this feature, we can provide a *hinted* lookup method with an amortized constant run-time.

Hinted Data Structures. The following code, shows an alternative implementation for set intersection that uses such hinted lookup operations:

```
hinted_set<K> result;
hinted_set<K>::hint_t hint = result.begin();
for(auto& e : set1) {
    hinted_set<K>::hint_t hint2 = set2.seek(e);
    if(hint2.found)
        hint = result.insert_hint(hint, e);
    set2.after(hint2);
}
```

The above *hinted set* data-structure enables faster insertion and lookup by providing a cursor through a *hint object* (of type `hint_t`). The `seek` method returns the hint object `hint2` pointing to element `e`. Thanks to the invocation of `set2.after(hint2)`, the irrelevant elements of `set2` (which are smaller than `e`) are no longer considered in the next iterations. The expression `hint2.found` specifies if the element exists in `set2` or not. Finally, if an element exists in the second set (specified by `hint2.found`), it is inserted into its correct position using `insert_hint`.

This paper introduces *hinted dictionaries*, a class of functional ordered data structures. The essential building block of hinted dictionaries are *hint objects*, that enable faster operations (than the traditional $O(\log n)$ complexity) by maintaining a pointer into the data structure.

Related Work. The existing work on efficient ordered dictionaries can be divided into two categories. First, in the imperative world, there are C++ ordered dictionaries (e.g., `std::map`) with *limited* hinting capabilities only for insertion through `emplace_hint`, but not for deletion and lookup, as observed previously.

¹ https://www.cplusplus.com/reference/set/set/emplace_hint/

Second, from the functional world, Adams' sets [1] provide efficient implementations for set-set operators. Functional languages such as Haskell have implemented ordered sets and maps based on them for more than twenty years [15]. Furthermore, it has been shown [4] that Adams' maps can be used to provide a parallel implementation for balanced trees such as AVL [2], Red-Black [3], Weight-Balanced [9], and Treaps [11]. However, Adams' maps do not expose any hint-based operations to the programmer. At first glance, these two approaches seem completely irrelevant to each other.

Contributions. *The key contribution of this paper is hinted dictionaries, an ordered data structure that unifies the techniques from both imperative and functional worlds.* The underlying representation for hinted dictionaries can be sorted arrays, unbalanced trees, and balanced trees by sharing the same interface. In our running example, alternative data-structure implementations can be easily provided by simply changing the type signature of the hinted set from `hinted_set` to another implementation, without modifying anything else.

This paper is organized as follows:

- We present monoid dictionaries, the most general form of dictionaries without any orderedness constraint on keys (Section 2). Such dictionaries subsume sets and maps and provide a restricted form of iterations in the form of map-reduce for computing associative and commutative aggregations over them (Section 2.4).
- Afterwards, we show ordered dictionaries, a more restricted class of dictionaries where the keys need to be ordered (Section 3). The iterations over these dictionaries are more general than monoid dictionaries, by relaxing the commutative requirement and providing associative aggregations. We show two particular interfaces for implementing associative aggregations in Section 3.3.
- We introduce hinted dictionaries, an implementation technique for ordered dictionaries (Section 4). A key ingredient of hinted dictionaries are the hint objects. In Section 4.2 we show operations over hint objects.
- Hinted dictionaries provide both sequential and parallel implementations for associative aggregations (Section 5).
- In order to support binary operations over dictionaries, hinted dictionaries provide a bulk operation interface (Section 6). The design choice for these bulk operations results in a completely different instantiation of hinted dictionaries. We present two implementations in this paper: insert-based (Section 6.1) and join-based (Section 6.2) hinted dictionaries.
- We present the implementation for hint objects in Section 7. More specifically, we present *focus-based hints*, hint objects focusing on a particular position in the dictionary (Section 7.1), and the corresponding focus-based hinted dictionaries (Section 7.2).
- We present the concrete implementations for hinted dictionaries in Section 8. More specifically, we show the implementation of ordered dictionaries using sorted arrays (Section 8.1), unbalanced trees (Section 8.2), and balanced trees (Section 8.3). Then, we connect all the components together in Section 8.4.
- Finally, we provide a C++ prototype for hinted dictionaries and discuss the challenges for tuning its performance in Section 9.1. We compare its performance with the standard library of C++ for set-set and sparse vector operations and show its asymptotic improvements in Section 9.2. For the latter workload we show its competitive performance with SciPy.

Throughout the paper, we use Scala to present the different components. The main ideas behind hinted dictionaries, however, can be implemented in other object-oriented and functional languages with support for generic types and lambda expressions (e.g., Haskell,

```

trait Monoid[V] {
  def op(e1: V, e2: V): V
  def zero: V
}

trait Equalable[K] {
  def equiv(e1: K, e2: K): Boolean
}

```

■ **Figure 1** The interface for monoid and equalable type classes.

OCaml, Java, Julia, Rust, and C++), as demonstrated in Section 9.1. The former is required for implementing the structural interfaces (e.g., `Monoid[T]`) and generic dictionaries (e.g., `Dict[K, V]`), while the latter is essential for implementing higher-order functions (e.g., `mapReduce`) in hinted dictionaries.

2 Monoid Dictionary

In this section, we present the most general form of dictionaries that we support; the ones where the values form a monoid structure, referred to as monoid dictionaries. We start by defining monoid and equalable values. Afterwards, we introduce the interface for monoid dictionaries. Finally, we show the class of iterations that can be expressed over them.

2.1 Monoid

A monoid is defined as a set of values V , with a binary operator `op` and a `zero` element, such that the following properties hold:

- **Associativity:** For all elements a, b, c in V : $\text{op}(\text{op}(a, b), c) = \text{op}(a, \text{op}(b, c))$
- **Identity Element:** For all elements a in V : $\text{op}(a, \text{zero}) = \text{op}(\text{zero}, a) = a$

An important class of monoids, supports the following additional axiom:

- **Commutativity:** For all elements a, b in V : $\text{op}(a, b) = \text{op}(b, a)$

Such monoids are referred to as *commutative monoids*. Important examples of commutative monoids are boolean values under conjunction and disjunction, integer numbers under multiplication and addition. Matrices of real numbers are commutative monoids under addition, but are non-commutative monoids under multiplication.

The Scala interface for monoid structures is shown in Figure 1. This interface can be thought of as a type class, where providing concrete implementations for this interface results in type class instances.

2.2 Equalable

In order to perform lookups over dictionaries, one requires to check for the equality of keys. This is achieved by the `Equalable` type class (Figure 1). Each type class instance provides the implementation strategy for checking the equivalence between two keys by overriding the `equiv` method.

2.3 Dictionary Interface

Given the key type K with an `Equalable[K]` constraint, and the value type V with a `Monoid[V]` constraint, one can define the interface `Dict[K, V, D]` for a dictionary type D (Figure 2).

```

trait Dict[K, V, D] {
  implicit val equ: Equalable[K]
  implicit val mon: Monoid[V]
  def find(dict: D, key: K): V
  def insert(dict: D, key: K, value: V): D
  def delete(dict: D, key: K): D
  def size(dict: D): Int
  def count(dict: D): Int
  def empty(): D
  def isEmpty(dict: D): Boolean
  final def single(key: K, value: V): D = insert(empty(), key, value)
}

```

■ **Figure 2** The interface for (monoid) dictionaries.

The specification of the methods of monoid dictionaries is as follows:

- **find**: performs a look up for the associated value with **key** in the dictionary **dict**. If the key does not exist in the dictionary, the identity element of the monoid structure over **V** is returned (**mon.zero**).
- **insert**: first performs a look up for the associated value with **key**. If the key does not exist, the pair of **key** and **value** is inserted in the dictionary. If the key already exists, the associated value, **old_value**, is updated by applying the binary operator of monoid to **old_value** and **value** (**mon.op(old_value, value)**). As the result, the updated dictionary is returned.
- **delete**: returns a new dictionary where **key** and its associated value is removed.
- **size**: returns the number of key-value pairs in the dictionary.
- **count**: returns the number of key-value pairs with non-zero values in the dictionary.
- **empty**: returns an empty dictionary of type **D** with keys and values of type **K** and **V**.
- **isEmpty**: check if the given dictionary is empty or not.
- **single**: returns a singleton dictionary containing the pair of **key** and **value**. This can be implemented by inserting into an empty dictionary.

By providing appropriate monoid structures for values, one can instantiate different collections from monoid dictionaries. As an example, using boolean values results in sets, using natural numbers results in bags, and using **Option** types results in maps.²

2.4 Iterations over Dictionaries

Next, we introduce the constructs required for performing iterations over dictionaries. As monoid dictionaries do not enforce any order over the keys, the iterative computation over them must be order-agnostic. Otherwise, the iterative computation results in different outcomes depending on the underlying organization of dictionary keys.

We provide the **mapReduce** method for expressing sound iterations over monoid dictionaries. This method performs map-reduce operations by starting from the initial element **z**, computing a transformation between key-value pairs to element of result type by **map**, and reducing the result elements by **red**. To ensure the soundness of aggregate computations, the **red** binary operator must be both commutative and associative.

² Using **Option** types incurs boxing and unboxing costs that is avoided by libraries such as **scala-unboxed-option** [5] for Scala and **unpacked sums** in GHC for Haskell [8]

```

trait DictIteration[K, V, D] { this: Dict[K, V, D] =>
  // precondition: 'red' must be commutative and associative
  def mapReduce[R](dict: D, z: R, map: (K, V) => R, red: (R, R) => R): R
  // precondition: 'R' must form a commutative monoid
  def aggregate[R: Monoid](dict: D, map: (K, V) => R): R = {
    val monR = implicitly[Monoid[R]]
    mapReduce[R](dict, monR.zero, map, monR.op)
  }
  def size(dict: D): Int =
    aggregate[Int](dict, (k, v) => 1)
  def count(dict: D): Int =
    aggregate[Int](dict, (k, v) => if(v == monR.zero) 0 else 1)
}

```

■ **Figure 3** The interface for iterations on dictionaries.

As an alternative interface, we provide `aggregate` with a monoid constraint over the result type. This method can be implemented in terms of `mapReduce` (cf. Figure 3). To do so, we need to use the zero element and the binary operator of an instance of the type class `Monoid[R]`. The Scala type system can provide an instance for type class `T` by using `implicitly[T]` [10]. In this case, `implicitly[Monoid[R]]` returns an instance of type `Monoid[R]`, where its zero element (`monR.zero`) and binary operator (`monR.op`) are passed as the initial value and reduction functions of `mapReduce`, respectively. Note that for sound aggregations, the result type should form a commutative monoid.

One can easily provide an implementation for `size` using the `aggregate` method. As we are only interested in counting the number of key-value pairs, it is sufficient to transform them to 1. Note that `size` returns the number of all pairs in the dictionary, while `count` only returns the number of pairs containing non-zero values.

For a cleaner presentation, we use the `DictIteration` interface to define the iteration-based methods (cf. Figure 3). This way, we avoid a large interface for `Dict`. To do so, we need to make sure that all classes and interfaces that extend `DictIteration`, also extend the `Dict` interface. This is achieved by ascribing the type of `this` object of the `DictIteration` interfaces with `Dict[K, V, D]`. Such dependency injection technique is known as *cake pattern* in the Scala programming language. It is important to note that using this technique is not essential; we can implement this code in a language without this feature by removing the interface for `DictIteration` altogether. Instead, all the method definitions of `DictIteration` are added to `Dict`.

3 Ordered Dictionary

In this section we present ordered dictionaries, the keys of which should follow a total order. First, we define the required interface for orderable keys in Section 3.1. Then, we introduce the interface for ordered dictionaries including bulk operations of them in Section 3.2. Finally, similar to monoid dictionaries, we show the class of iterations expressible over ordered dictionaries in Section 3.3.

3.1 Orderable

In this section, we introduce the interface required for the keys of ordered dictionaries (cf. Figure 4). In ordered dictionaries, apart from the need to check for the equality of two keys (using `equiv` derived from `Equalable`), a total order must also be provided.

```

trait Orderable[K] extends Equalable[K] {
  def compare(e1: K, e2: K): Int
  def max: K
  def min: K
  final def lt(e1: K, e2: K): Boolean = compare(e1, e2) < 0
  final def gt(e1: K, e2: K): Boolean = compare(e1, e2) > 0
  final def lteq(e1: K, e2: K): Boolean = compare(e1, e2) <= 0
  final def gteq(e1: K, e2: K): Boolean = compare(e1, e2) >= 0
  final def equiv(e1: K, e2: K): Boolean = compare(e1, e2) == 0
}

```

■ **Figure 4** The interface for orderable.

The `compare` method is sufficient to provide the total order information. If its return value is a positive number, the first element is greater than the second value, and for a negative number, vice versa. Otherwise, if the return value is zero, this means that both elements are equal. All comparison operators can be implemented using the `compare` method, as can be seen in Figure 4.

As we are only interested in finite dictionaries, one can provide an upper bound and lower bound for keys. These values are specified using `max` and `min` in the `Orderable` interface. We will see in Section 7.1 how upper bounds can be used for implementing hint objects.

3.2 Ordered Dictionary Interface

The interface of ordered dictionaries is very similar to monoid dictionaries. The `Equalable` type class instance is the same as the one used for `Orderable`. This is because the `Orderable` interface subsumes the interface of `Equalable` by using inheritance.

The additional methods provided for ordered dictionaries are as follows:

- `toList`: this method converts ordered dictionaries into a list of type `List[(K, V)]`.³
- `append`: for two ordered dictionaries `left` and `right`, where all the keys of `left` are less than the keys of `right`, this method returns an ordered dictionary containing the elements of both dictionaries.
- `join`: given two ordered dictionaries `left` and `right` and a key-value pair `key` and `value`, this method creates an ordered dictionary containing the elements of `left` and `right` as well as the pair of `key` and `value`. In this method, all the keys of `left` must be less than `key`, and all the keys of `right` must be more than `key`.

Note that the mentioned preconditions for the last two methods are necessary to preserve the dictionary's order, and violating any of them makes hinted dictionaries not work. Furthermore, these two methods are bulk operations and thus are defined in a separate `OrderedDictBulkOps` interface (Figure 5) following the cake pattern. These methods are critical for providing different concrete ordered dictionary implementations, as will be observed in Section 6.

3.3 Iterations over Ordered Dictionaries

As opposed to monoid dictionaries, ordered dictionaries do not need the reduction function to be commutative. This is because even with non-commutative reductions, ordered dictionaries will result in a deterministic order for key-value pairs.

³ Unordered dictionaries cannot implement `toList` as there is no deterministic order for the key-value pairs.

```

trait OrderedDict[K, V, D] extends Dict[K, V, D] {
  implicit val ord: Orderable[K]
  implicit val equ: Equalable[K] = ord
  def toList(dict: D): List[(K, V)]
}

trait OrderedDictBulkOps[K, V, D] { this: OrderedDict[K, V, D] =>
  // precondition: keys(left) < keys(right)
  def append(left: D, right: D): D
  // precondition: keys(left) < k < keys(right)
  def join(left: D, key: K, value: V, right: D): D
}

```

■ **Figure 5** The interface for ordered dictionaries and bulk operations over them.

The `toList` method can be implemented by `mapReduce` and `aggregate` methods. Figure 6 shows its implementation using `aggregate`; it is sufficient to map each of the key-value pairs into a singleton list. This requires the following instance of `Monoid[List[T]]`:

```

implicit def MonoidList[T] = new Monoid[List[T]] {
  def op(e1: List[T], e2: List[T]): List[T] = e1 ++ e2
  def zero: List[T] = Nil
}

```

Here, the binary operator is list append (`++`) and the zero element is the empty list (`Nil`). The `toList` method returns the result of appending all these singleton lists.

3.3.1 Sequential Iterations

Similar to functional lists in functional languages, ordered dictionaries also provide a `foldLeft` method for performing accumulating computations over their elements from left to right. This method is provided in the `OrderedDictFoldLeft` interface (cf. Figure 6).

The `mapReduce` method can be implemented using `foldLeft` by passing the initial value and defining the accumulating function as applying the `red` function to the previous state `s` and the result of `map(k, v)`.

3.3.2 Parallel Iterations

Thanks to the associative nature of reduction functions, there is no need to perform aggregations only sequentially from left to right. Instead, one can perform aggregations in a tree-structured manner, which is more parallel-friendly.

The `foldTree` method provided in the `OrderedDictFoldTree` interface (cf. Figure 6) is responsible for computing parallel iterations. This is provided by performing a top down traversal over the logical tree representation. Similar to `foldLeft`, this method accepts an initial state (`z`). At each stage, it computes the stage to be passed to each of the subtrees. The `op` method produces a triple of elements when applied to the current key-value pair and the previous state. The first two elements of this triple are the states to be passed to each of subtrees. The last element corresponds to a hidden state of type `M`. This hidden state is used after the aggregation for subtrees are computed. The `comb` method applies this hidden state alongside with the current key-value and the states return by the subtrees and computes the next state.


```

trait OrderedDictIteration[K, V, D] extends DictIteration[K, V, D] {
  this: OrderedDict[K, V, D] =>
  // precondition: 'red' should only be associative
  def mapReduce[R](dict: D, z: R, map: (K, V) => R, red: (R, R) => R): R
  def toList(dict: D): List[(K, V)] =
    aggregate[List[(K, V)]](dict, (k, v) => List((k, v)))
}

trait OrderedDictFoldLeft[K, V, D] extends OrderedDictIteration[K, V, D] {
  this: OrderedDict[K, V, D] =>
  def foldLeft[R](dict: D, z: R, op: (R, K, V) => R): R
  override def mapReduce[R](dict: D, z: R, map: (K, V) => R, red: (R, R) => R): R =
    foldLeft[R](dict, z, (s, k, v) => red(s, map(k, v)))
}

trait OrderedDictFoldTree[K, V, D] extends OrderedDictIteration[K, V, D] {
  this: OrderedDict[K, V, D] =>
  def foldTree[R, M](dict: D, z: R, op: (K, V, R) => (R, R, M),
    comb: (K, V, M, R, R) => R): R
  override def mapReduce[R](dict: D, z: R, map: (K, V) => R, red: (R, R) => R): R =
    foldTree[R, Unit](dict, z, (k, v, s) => (s, s, ()),
      (k, v, _, s1, s2) => red(red(s1, map(k, v)), s2))
}

```

■ **Figure 6** The interface for iterations over ordered dictionaries. The `foldLeft` method corresponds to computing aggregations sequentially and `foldTree` method is a parallel-friendly aggregate computation strategy.

The `mapReduce` method can be implemented using `foldTree` as well. As the `op` method, we return the previous state `s` to both subtrees. As the `comb` method, we apply the reduction method twice. The first application involves the return state of left subtree (`s1`) and the mapping of key-value pair `map(k, v)`. The second application is over the result of the previous reduction and the state of the right subtree (`s2`).

Note that for implementing `mapReduce` there was no hidden state required, and thus the unit value `()` with unit type `Unit` was provided. We will see cases where this hidden state will be required in Section 5.2.1.

By carefully keeping the value of aggregation in the ordered dictionary, one can provide a more efficient implementation for `mapReduce`. This is similar to the idea of augmented trees, and has already been investigated in Augmented Maps [17].

4 Hinted Dictionary

In this section, we introduce hinted dictionaries, an implementation strategy for ordered dictionaries. First, we present the interface of hinted dictionaries in Section 4.1. Then, we show the interface for hint objects in Section 4.2.

4.1 Hinted Dictionary Interface

Hinted dictionaries inherit all the methods of both monoid dictionaries and ordered dictionaries. Additionally, they provide the following methods:

- **begin**: returns the hint object corresponding to the beginning of dictionary. This method is useful for accessing the head of an ordered dictionary.

28:10 Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

```
trait HintedDict[K, V, D, H] extends OrderedDict[K, V, D] {
  def begin(dict: D): H
  def middle(dict: D): H
  def end(dict: D): H
  def isEnd(dict: D, hint: H): Boolean
  def next(dict: D, hint: H): H
  def seek(dict: D, key: K): H
  // precondition: current(hint)._1 == key
  def findHint(dict: D, hint: H, key: K): V
  // precondition: current(hint)._1 == key
  def insertHint(dict: D, hint: H, key: K, value: V): D
  // precondition: current(hint)._1 == key
  def deleteHint(dict: D, hint: H, key: K): D
  def insert(dict: D, key: K, value: V): D =
    insertHint(dict, seek(dict, key), key, value)
  def delete(dict: D, key: K): D =
    deleteHint(dict, seek(dict, key), key)
  def find(dict: D, key: K): V =
    findHint(dict, seek(dict, key), key)
}
```

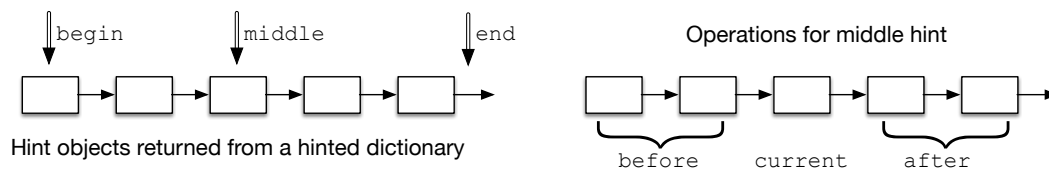
■ **Figure 7** The interface for hinted dictionaries.

- **middle**: returns the hint object of the middle of the dictionary. This method is useful for cases that require viewing the dictionary as a tree. For example, it can be used for a binary search where one needs to access the middle of a collection.
- **end**: returns the hint object specifying the end of dictionaries.
- **isEnd**: checks whether the given hint object corresponds to the end of the dictionary. This is useful for terminating an iteration over the hinted dictionary.
- **next**: returns the hint object succeeding the given hint object over the input dictionary.
- **seek**: returns the hint object for the position in the array where **key** would be placed. This means that the preceding elements have smaller keys and succeeding elements have larger keys. In the case that the dictionary contains the **key**, the corresponding hint object is returned.
- **findHint**: returns the associated value with the given key using the provided hint object. As the precondition, the hint should point to the correct position.
- **insertHint**: inserts the given key-value pair to the position provided by the hint object. Similar to the previous method, the hint object assumed to point to the correct position.
- **deleteHint**: deletes the key-value pair corresponding to the input key using the provided hint object. A similar precondition to the previous two methods hold.

By using the **seek** method to compute the correct hint object, one can have an implementation for **find**, **insert**, and **delete** using the corresponding hinted methods. Once hinted dictionaries are supplied with (amortized) constant-time operations for these hinted methods, one can better benefit from their efficiency.

4.2 Hint Operations

The hint objects can be thought as pointers to different locations of an ordered dictionary (cf. Figure 8). Rather than providing a concrete implementation for hint object, we provide an interface for them in Figure 9. We leave the actual implementation for these operators to Section 7.



■ **Figure 8** An example hinted dictionary representation showing the hint objects returned by `begin`, `middle`, and `end` methods and the outcome of `before`, `current`, and `after` over the hint object returned by `middle`.

```

trait HintOps[K, V, D, H] { this: HintedDict[K, V, D, H] =>
  def before(hint: H): D
  def after(hint: H): D
  def current(hint: H): (K, V)
}

```

■ **Figure 9** The interface for operations on hint objects.

The methods for hint objects are as follows:

- **before**: returns the dictionary of elements located before the hint object.
- **after**: returns the dictionary of elements that are after the hint object.
- **current**: returns the key-value pair of the entry of the dictionary that hint object is pointing into. If for a given key, there is no associated value, the identity element of monoid is returned, similar to what was observed for `find`.

As it was shown in Figure 7 we can use these methods to provide preconditions for the hinted dictionary methods. In addition, all the implementations of the `seek` method need to enforce the following post-condition. Assuming the result hint object is `res`, the key of this element should be the same as the input key (`current(res)._1 == key`).⁴ Furthermore, the keys of the dictionary before the result hint object (`before(res)`) should be less than `key`. Similarly, the keys of the dictionary after the result hint object (`after(res)`) should be greater than `key`.

5 Iterations

In this section, we use the methods provided by hinted dictionaries to implement sequential and parallel iterations over them.

5.1 Sequential Implementation

The interface of `HintedDictFoldLeft` containing the implementation for `foldLeft` is presented in Figure 10. Recall that this method is useful for stateful iterations over dictionaries. The initial state is specified by `z`, and the state is updated by applying the function `op` to the current state and key-value pairs.

In order to implement `foldLeft`, we define a recursive function `foldLeftTR` inside it. This function has two input parameters `hint` and `res`, which specify the current hint object and the computed state by iterating up to that hint object. We start by passing the initial

⁴ Note that tuple indexing in Scala starts from 1, and `tup._i` where `i` is an integer shows the `i`th element of the tuple.

28:12 Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

```
trait HintedDictFoldLeft[K, V, D, H] extends OrderedDictFoldLeft[K, V, D]
  with HintedDict[K, V, D, H] with HintOps[K, V, D, H] {
  def foldLeft[R](dict: D, z: R, op: (R, K, V) => R): R = {
    @tailrec def foldLeftTR(hint: H, res: R): R =
      if(!isEnd(dict, hint)){
        val (key, value) = current(hint)
        val next_res = op(res, key, value)
        val next_hint = next(dict, hint)
        foldLeftTR(next_hint, next_res)
      }
      else
        res
    foldLeftTR(begin(dict), z)
  }
}
```

■ **Figure 10** The implementation of sequential (fold-left-based) iterations over hinted dictionaries.

state `z` as the value of `res`, and the beginning of the dictionary as the value of `hint`. The function `foldLeftTR` is recursively called until the hint object does not point to the end of dictionary (`!isEnd(dict, hint)`). At each recursive call, we compute the next state value (`next_res`) by applying `op` to the current state (`res`) and key-value pair and the next hint object (`next_hint`) by `next(dict, hint)`.

Note that the definition of `foldLeftTR` is annotated with `@tailrec`. This means that this function is tail recursive – all recursive calls are appearing as the last statement. This annotation ensures that the Scala compiler can turn this method into imperative `while` loops, which results in better performance by removing the need to increase the stack frame size.

5.1.1 Example: Sparse Vector Inner Product

Figure 11 shows the interface for sparse vectors. A sparse vector `Vec` can be represented as a dictionary from indices to a scalar value `Sca`. In order to support operation such as inner product that involve multiplication over scalar values, we need to define a type class instance for monoid over scalar values under multiplication (specified by `prod`).

An efficient sequential implementation of the inner product is provided in Figure 11. The `foldLeft` method accepts a pair of states containing the result of inner product as well as the rest of the second vector to process. The state is initially set to the monoid identity element (`dict.mon.zero`) and the second vector (`v2`). At each iteration, the `seek` method for the given key is invoked over the remaining part of the second vector. Then, the result of inner product is updated by adding the previous result (`res`) to the outcome of multiplying (`prod.op`) the current value (`v`) with the value specified by the hint object (`current(hint)._2`). Note that in the case that the second vector does not have any elements at index `k`, the specified value by its hint object would be zero (`dict.mon.zero`). Finally, the rest of the second vector is computed by taking only the elements of dictionary happening after the hint object (`after(hint)`).

5.2 Parallel Implementation

The `foldTree` method can also be used for performing stateful iterations over hinted dictionaries. As opposed to `foldLeft`, this method can perform the computation in a divide-and-conquer manner. Figure 12 demonstrates the process of applying `foldTree` over a

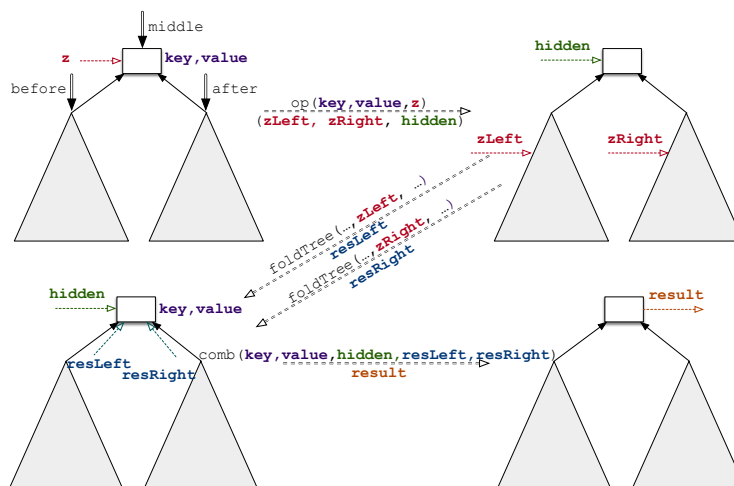
```

trait SparseVectorOps[Sca, Vec] {
  implicit val prod: Monoid[Sca]
  def inner(v1: Vec, v2: Vec): Sca
}

trait SparseVectorFoldLeftOps[Sca, Vec, H] extends SparseVectorOps[Sca, Vec] {
  val dict: HintedDict[Int, Sca, Vec, H]
  val dictFolding: OrderedDictFoldLeft[Int, Sca, Vec]
  val hintOps: HintOps[Int, Sca, Vec, H]
  import dict._
  import dictFolding._
  import hintOps._
  def inner(v1: Vec, v2: Vec): Sca =
    foldLeft[(Sca, Vec)](v1, (dict.mon.zero, v2), (s, k, v) => {
      val (res, v2p) = s
      val hint = seek(v2p, k)
      dict.mon.op(res, prod.op(v, current(hint)._2)) -> after(hint)
    })._1
}

```

■ **Figure 11** The implementation of the inner product of two sparse vectors using sequential iteration.



■ **Figure 12** The process of executing `foldTree` on a logical tree view of an ordered dictionary. The recursive invocations of `foldTree` on the two sub-trees can be evaluated in parallel.

hinted dictionary viewed as a tree. The divide phase involves recursively applying `foldTree` to the left and right subtrees, where the initial state for each recursive call is computed using the function `op`. The conquer phase uses the function `comb` to combine the results of recursive calls to compute the output state of the entire tree.

The interface of `HintedDictFoldTree` provides the implementation for `foldTree` using hinted dictionaries (cf. Figure 13). If the input dictionary is empty the initial value is returned. Otherwise, the following steps are performed. First, the hint object returned by `middle` is used to retrieve the key-value pairs in the middle of the dictionary. The `op` function is applied to this key-value pair and the previous state `z`. This results in the initial state of the left and right sub-trees (`zLeft` and `zRight`) as well as the hidden state (`hidden`). The

28:14 Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

```
trait HintedDictFoldTree[K, V, D, H] extends OrderedDictFoldTree[K, V, D]
  with HintedDict[K, V, D, H] with HintOps[K, V, D, H] {
  def foldTree[R, M](dict: D, z: R, op: (K, V, R) => (R, R, M),
    comb: (K, V, M, R, R) => R): R =
    if(isEmpty(dict)) z
    else {
      val hint = middle(dict)
      val (key, value) = current(hint)
      val (zLeft, zRight, hidden) = op(key, value, z)
      val resLeft = foldTree(before(hint), zLeft, op, comb)
      val resRight = foldTree(after(hint), zRight, op, comb)
      comb(key, value, hidden, resLeft, resRight)
    }
}
```

■ **Figure 13** The implementation of parallel (fold-tree-based) iterations over hinted dictionaries.

`foldTree` method is recursively invoked for both left and right sub-trees (`before(hint)` and `after(hint)`) using their corresponding initial states. These two invocations are independent of each other and can be run in parallel. Finally, the key-value pair, the hidden state, and the results of recursive calls (`resLeft` and `resRight`) are combined by applying the `comb` function.

Next, we show example usages of these iteration constructs.

5.2.1 Example: Set-Set Union

A set of type `S`, consisting elements of type `K` can be expressed as a dictionary with keys of type `K` and values of type `Boolean`. Hence, they can also be expressed using hinted dictionaries.

Figure 14 provides the interface for set-set operations such as `union`, `intersect`, and `difference`. For the sake of brevity here we only show the implementation for `union` using `foldTree`. The `foldLeft`-based implementation and the one for `intersect` and `difference` can be similarly obtained.

The result type of `foldTree` (cf. Figure 14) is the set type `S`, and the type of its hidden state is the hint type `H`. The iteration is over `set1` with the initial state of `set2`. At each stage, the `seek` method looks for the key `k` in the set specified by its current state `s`. As the state for the left and right sub-trees, the dictionaries with preceding and succeeding elements (`before(hint)` and `after(hint)`) are provided, and as the hidden state the hint object `hint` is returned. For the combine operation, the state returned by the left sub-tree (`s1`), the current key-value pair with update value (`k` and `mon.op(v, current(hidden)._2)`), and the state returned by the right sub-tree (`s2`) are joined.

6 Bulk Operations

In this section, we provide two design decisions for implementing bulk operations. The first design is insert-based hinted dictionaries, where the bulk operations are derived from the implementation for the hinted insertion method (Section 6.1). The second design revolves around using the `join` method as the central operator (Section 6.2).

```

trait SetSetOps[S] {
  def union(set1: S, set2: S): S
  def intersect(set1: S, set2: S): S
  def difference(set1: S, set2: S): S
}

trait SetSetFoldTreeOps[K, S, H] extends SetSetOps[S] {
  val dict: HintedDict[K, Boolean, S, H]
  val dictFolding: OrderedDictFoldTree[K, Boolean, S]
  val hintOps: HintOps[K, Boolean, S, H]
  val dictDictOps: OrderedDictBulkOps[K, Boolean, S]
  import dict._
  import dictFolding._
  import hintOps._
  import dictDictOps._

  def union(set1: S, set2: S): S =
    foldTree[S, H](set1, set2, (k, v, s) => {
      val hint = seek(s, k)
      (before(hint), after(hint), hint)
    }, (k, v, hidden, s1, s2) => join(s1, k, mon.op(v, current(hidden)._2), s2))
    // ...
}

```

■ **Figure 14** The implementation of set-set union using fold-tree-based iteration.

6.1 Insert-Based Hinted Dictionaries

The hinted insertion method (`insertHint`) is an expressive operation. By providing a concrete implementation for this method, one can guide how an ordered dictionary can be inductively constructed starting from an empty dictionary.

Figure 15 shows the implementation of insert-based hinted dictionaries. The `insertHint` method is left as abstract. Providing different implementations result in a completely different strategy for maintaining ordered dictionaries.

The `append` method is implemented by iterating over the second dictionary and adding its elements one-by-one to the first dictionary. This is achieved by `foldLeft` over `right`, with an initial state of `left`, and adding the key-value pairs of `right` to the end of the previously computed result. This implementation is correct because we know that the keys appearing in the second dictionary are greater than the keys of the first dictionary. By assuming that we provide an implementation for `insertHint` with an amortized constant-time complexity, the `append` operation will have an amortized linear run-time complexity.

The `join` method can be implemented in terms of `append` and `insertHint`. First, we need to insert the given key-value pair to the end of the first dictionary. Afterwards, this intermediate dictionary is appended by the second dictionary. By making the same assumptions as `append`, this operation has also an amortized linear run-time complexity.

6.2 Join-Based Hinted Dictionaries

An alternative way for defining hinted dictionaries is based on the join operator. This is inspired by Adams' sets [1] and the follow up parallel implementations [4].

```

trait InsertBasedDict[K, V, D, H] extends HintedDict[K, V, D, H]
  with HintOps[K, V, D, H] with OrderedDictBulkOps[K, V, D]
  with HintedDictFoldLeft[K, V, D, H] {
  // precondition: keys(left) < keys(right)
  def append(left: D, right: D): D =
    foldLeft[D](right, left, (s, k, v) =>
      insertHint(s, end(s), k, v)
    )
  // precondition: keys(left) < key < keys(right)
  def join(left: D, key: K, value: V, right: D): D =
    append(insertHint(left, end(left), key, value), right)
  // precondition: current(hint)._1 == key
  def insertHint(dict: D, hint: H, key: K, value: V): D
}

```

■ **Figure 15** The implementation of insert-based hinted dictionaries.

The implementation of join-based hinted dictionaries is shown in Figure 16. The `join` method does not have a concrete implementation. It has been shown [4] that different balanced tree representations such as AVL [2], Red-Black [3], Weight-Balanced [9], and Treaps [11] can be expressed by providing an appropriate implementation for the `join` method.

The `append` method is expressed as follows. If the second dictionary is empty, the first dictionary is returned. Otherwise, the hint object for the beginning of the second dictionary is used to retrieve its first key-value pair. Then, the `join` method is applied to the first dictionary, the first key-value pair, and the rest of the second dictionary. If the `join` method is an amortized linear operation, the `append` method also follows the same run-time complexity.

The `insertHint` method is expressed by joining the dictionary preceding the hint object (`before(hint)`), the key-value pair with updated value (`mon.op(current(hint)._2, value)`), and the dictionary succeeding the hint object (`after(hint)`). Note that this way of implementing `insertHint` is suboptimal given that the `join` is a linear time operator. Thus, one has to try avoid using `insertHint` for join-based hinted dictionaries for performance reasons.

The efficiency of hinted dictionaries is not solely dependent on efficient `join` and `insertHint` operations. Having an efficient hint object implementation is also essential, which will be presented next.

7 Hint Implementation

This section starts with a concrete representation for hint objects. Using this representation we provide the implementation for hint operations in Section 7.1. Afterwards, we provide the implementation of the rest of the methods of hinted dictionaries in Section 7.2.

7.1 Focus-Based Hints

As it was stated in Section 4.2, hint objects can be viewed as pointers to different locations of an ordered dictionary. In this section, we consider them as objects focusing on a particular position in the dictionary. The key-value pair that the hint object focuses on, specifies the `key` and `value` fields of the `FocusHint` class. The lack of a key-value pair is specified by putting the identity element of the underlying monoid for type `V`. The sub-dictionary containing the elements preceding/succeeding the focused key-value pair are stored in `left/right`.


```

trait JoinBasedDict[K, V, D, H] extends HintedDict[K, V, D, H]
  with HintOps[K, V, D, H] with OrderedDictBulkOps[K, V, D] {
  // precondition: keys(left) < keys(right)
  def append(left: D, right: D): D = {
    if (isEmpty(right))
      left
    else {
      val hint = begin(right)
      val (key, value) = current(hint)
      val rightNew = after(hint)
      join(left, key, value, rightNew)
    }
  }
  // precondition: keys(left) < key < keys(right)
  def join(left: D, key: K, value: V, right: D): D
  // precondition: current(hint)._1 == key
  def insertHint(dict: D, hint: H, key: K, value: V): D =
    join(before(hint), key, mon.op(current(hint)._2, value), after(hint))
}

```

■ **Figure 16** The implementation of join-based hinted dictionaries.

```

case class FocusHint[K, V, D](left: D, key: K, value: V, right: D)

trait FocusHintOps[K, V, D] extends HintOps[K, V, D, FocusHint[K, V, D]] {
  this: HintedDict[K, V, D, FocusHint[K, V, D]] =>
  type H = FocusHint[K, V, D]
  def before(hint: H): D = hint.left
  def after(hint: H): D = hint.right
  def current(hint: H): (K, V) = (hint.key, hint.value)
}

```

■ **Figure 17** The implementation for focus hint and its operations.

The implementation of the hint operations using focus-based hints is very natural: `before` and `after` return `left` and `right` fields of the `FocusHint` object, and `current` returns the pair `(hint.key, hint.value)`.

7.2 Focus-Based Hinted Dictionaries

Figure 18 shows the implementation of focus-based hinted dictionaries, where the hint objects are `FocusHints`. The following methods are left as abstract: `empty`, `isEmpty`, `begin`, and `middle`. Depending on the underlying data structure, the implementation for these methods can be different.

The methods implemented by focus-based dictionaries are as follows:

- **seek**: If the given dictionary is empty, an empty `FocusHint` object is returned the key of which is the same as the input key. Otherwise, it performs a binary search to return an appropriate hint object. For binary search, the key of the middle of the dictionary is compared with the input key. If the middle key is the same as the input key, the middle hint object is returned. If the input key is less than the middle key, the process is recursively invoked for the preceding dictionary (`seek(1, key)`), and the result hint object is computed by substituting its right dictionary by joining it with the rest of the input dictionary. A similar process is performed when the input key is greater than the middle key.

28:18 Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

```
trait FocusHintedDict[K, V, D] extends HintedDict[K, V, D, FocusHint[K, V, D]]
  with FocusHintOps[K, V, D] with OrderedDictBulkOps[K, V, D] {
  // postcond: res.key == key
  def seek(dict: D, key: K): H = {
    if(isEmpty(dict))
      FocusHint(empty(), key, mon.zero, empty())
    else {
      val hint@FocusHint(l, m, v, r) = middle(dict)
      if(ord.equiv(key, m)) {
        hint
      } else if(ord.lt(key, m)) {
        val hint2 = seek(l, key)
        hint2.copy(right = join(hint2.right, m, v, r))
      } else {
        val hint2 = seek(r, key)
        hint2.copy(left = join(l, m, v, hint2.left))
      } } }
  def end(dict: D): H = FocusHint(dict, ord.max, mon.zero, empty())
  def isEnd(dict: D, hint: H): Boolean = hint == end(dict)
  def next(dict: D, hint: H): H = {
    val rightDict = after(hint)
    val nextHint = begin(rightDict)
    if(isEnd(rightDict, nextHint))
      end(dict)
    else {
      val (k, v) = current(nextHint)
      FocusHint(before(hint), k, v, after(nextHint))
    } }
  def deleteHint(dict: D, hint: H, k: K): D = append(hint.left, hint.right)
  def findHint(dict: D, hint: H, k: K): V = hint.value
}
```

■ **Figure 18** The implementation of focus-based hinted dictionary.

- **end**: Returns an empty `FocusHint` object with the key set to the upper bound of keys (`ord.max`).
- **isEnd**: Check if the given hint is the same as the end hint object.
- **next**: The hint object for the succeeding dictionary is constructed. If this hint object corresponds to the end of that dictionary, then the end of the input dictionary is returned. Otherwise, the return hint object is constructed by using this hint object and the preceding dictionary of the input dictionary.
- **deleteHint**: It is computed by appending the preceding and succeeding dictionaries together.
- **findHint**: The associated value with the hint object is returned. In the case where the key does not exist, the hint object stores the identity of the underlying monoid.

8 Concrete Implementations

In this section, provide three categories of concrete implementations for hinted dictionaries. We start by using sorted arrays as the underlying implementation in Section 8.1. Then, we show how unbalanced trees can be used for implementing hinted dictionaries in Section 8.2. Finally, we use balanced trees as the representation for hinted dictionaries in Section 8.3.

```

import scala.collection.mutable.ArrayBuffer

case class ArrayView[T](buffer: ArrayBuffer[T], lower: Int, upper: Int)

trait ArrayDict[K, V] extends FocusHintedDict[K, V, ArrayView[(K, V)]] {
  type D = ArrayView[(K, V)]
  def empty(): D = ArrayView(ArrayBuffer(), 0, 0)
  def isEmpty(dict: D): Boolean = dict.upper == dict.lower
  def begin(dict: D): H =
    if(isEmpty(dict)) end(dict)
    else {
      val (k, v) = dict.buffer(dict.lower)
      FocusHint(empty(), k, v, ArrayView(dict.buffer, dict.lower + 1, dict.upper))
    }
  def middle(dict: D): H = {
    if(isEmpty(dict)) end(dict)
    else {
      val mid = (dict.lower + dict.upper) / 2
      val (k, v) = dict.buffer(mid)
      val l = ArrayView(dict.buffer, dict.lower, mid)
      val r = ArrayView(dict.buffer, mid + 1, dict.upper)
      FocusHint(l, k, v, r)
    }
  }
}

class CopyingArrayDict[K, V](
  implicit override val ord: Orderable[K], val mon: Monoid[V]) extends
  ArrayDict[K, V]
  with InsertBasedDict[K, V, ArrayView[(K, V)], FocusHint[K, V, ArrayView[(K,
  V)]]] {
  def insertHint(dict: D, hint: H, key: K, value: V): D = {
    val array = dict.buffer.clone() // can be removed if dict is no longer used
    val idx = before(hint).upper
    val (prevKey, prevValue) = current(hint)
    val newUpper =
      if(prevValue == mon.zero) {
        array.insert(idx, (key, value))
        dict.upper + 1
      } else {
        array(idx) = (key, mon.op(prevValue, value))
        dict.upper
      }
    ArrayView(array, dict.lower, newUpper)
  }
}

```

■ **Figure 19** The implementation of hinted dictionaries using an underlying sorted array.

8.1 Sorted Array

Using sorted arrays is one of the main techniques for a sequential implementation of ordered dictionaries. In the C++ world, the `flat_map` container provided by the Boost library [14] uses sorted arrays for representing dictionaries. This data structure is particularly useful for the workloads where the insertions are applied to the end of the ordered dictionary.

Inspired by these C++ implementations, Figure 19 provides the implementation of hinted dictionaries using sorted arrays. In order to preserve an underlying array we use `ArrayBuffers`, mutable containers similar to `std::vectors` of C++. The `ArrayView` data type represents a subset of an `ArrayBuffer` bounded by the indices specified by `lower` and `upper`.

The implementations for `empty`, `isEmpty`, and `begin` are straightforward. For the `middle` method, we need to first retrieve the index of the middle element (`mid`). Then, we compute the preceding dictionary by using the same lower bound, but with the upper bound specified by `mid`. Similarly, the succeeding dictionary uses the lower bound specified by `mid+1`, but with the same upper bound. Finally, we return the focused hint object based on the key-value pair of the middle element and the preceding and succeeding dictionaries.

To create the left and right `ArrayViews` of the hint object, we can share the underlying buffer of the current `ArrayDict` without copying it. Since `ArrayView` is using `ArrayBuffer` as the underlying array and the start and end of the `ArrayView` are determined by `lower` and `upper`, changing these parameters can result in a new dictionary. This sharing opportunity frees the code from copying elements every time we are using `middle`. A similar opportunity is available for `before` and `after`.

Because of using arrays as the underlying representation, it would be more efficient to follow an `InsertBasedDict` interface. To implement the `insertHint` method, we need to check if the hint object points to an actual element. This is achieved by checking if the associated value is different than the identity element of the underlying monoid. In the case of the existence of an element with the same key, the size of the underlying `ArrayBuffer` does not need to change; it is sufficient to update the value of the existing element by applying the binary operator of the monoid to the previous value and the new value to be inserted (`mon.op(prevValue, value)`). If the hint object does not point to an actual element, we need to insert the given key-value pair in the specified position. Finally, we adjust the upper bound and return the updated array.

Note that in the case that there is no more references to the input dictionary in the user program, one can perform in-place update and there would be no need to copy the original array. We leave the implementation of the in-place update version for the sake of brevity.

8.2 Unbalanced trees

An alternative implementation for hinted dictionaries is based on a tree-based representation. We first give a generic implementation for tree-based hinted dictionaries that can be used for both unbalanced and balanced tree representations. Afterwards, we show a simple representation where no smart effort is invested for maintaining the tree in balance.

Figure 20 provides the generalized implementation for ordered dictionaries using trees. The `Tree` data type is defined as an ADT (Algebraic Data Type), where `Leaf` corresponds to a leaf and `Bin` specifies an intermediate node. As the tree nodes can maintain additional information (e.g., height for balanced trees), the type member `Entity` is used for keeping the type of the information kept by each tree node. The `key` and `value` methods are used to extract keys and values from the elements, respectively.

As opposed to sorted arrays, the tree-based hinted dictionaries have a straightforward implementation for `middle`; for leaves the hint object for `end` is returned, whereas for intermediate nodes the focused-hint object with 1) the left sub-tree as the preceding dictionary, 2) the right sub-tree as the succeeding dictionary, and 3) the key/value of its element as the key-value pair is returned. To implement `begin`, the helper method `seekFirst` is defined, which looks for the smallest key-value pair and returns them alongside the succeeding dictionary. These values are used to return the hint object with an empty preceding dictionary.

```
sealed trait Tree[T]
case class Bin[T](l: Tree[T], v: T, r: Tree[T]) extends Tree[T]
case class Leaf[T]() extends Tree[T]

trait TreeDict[K, V, E] extends FocusHintedDict[K, V, Tree[E]] {
  type Entity = E
  type D = Tree[E]
  def key(e: Entity): K
  def value(e: Entity): V
  def empty(): D = Leaf()
  def isEmpty(dict: D): Boolean = dict == Leaf()
  def seekFirst(dict: D): (D, K, V) = {
    val FocusHint(l, k, v, r) = middle(dict)
    if(isEmpty(l)) (r, k, v)
    else {
      val (tp, kp, vp) = seekFirst(l)
      (join(tp, k, v, r), kp, vp)
    }
  }
  def begin(dict: D): H =
    if(isEmpty(dict)) end(dict)
    else {
      val (r, k, v) = seekFirst(dict)
      FocusHint(empty(), k, v, r)
    }
  def middle(dict: D): H = dict match {
    case Leaf() => end(dict)
    case Bin(l, e, r) => FocusHint(l, key(e), value(e), r)
  }
}
```

■ **Figure 20** The generalized implementation for tree-based representations of hinted dictionaries.

```
class UnbalancedDict[K, V](implicit override val ord: Orderable[K], val mon:
  Monoid[V])
  extends TreeDict[K, V, (K, V)]
  with JoinBasedDict[K, V, Tree[(K, V)], FocusHint[K, V, Tree[(K, V)]]]
  with HintedDictFoldTree[K, V, Tree[(K, V)], FocusHint[K, V, Tree[(K, V)]]] {
  def key(e: Entity): K = e._1
  def value(e: Entity): V = e._2
  def join(left: D, key: K, value: V, right: D): D = Bin(left, (key, value), right)
}
```

■ **Figure 21** The implementation of hinted dictionaries using unbalanced binary trees.

Figure 21 shows the implementation for hinted dictionaries that use unbalanced trees. Because of the tree-based representation, a natural interface for `UnbalancedDict` is the join-based hinted dictionary, although one could use the insert-based one with worse performance. The tree nodes do not need to maintain any additional information. Thus, the entity type of the tree nodes is the key-value pair $((K, V))$. The implementation for the `join` is to simply create an intermediate node with the given key-value pair as the content, and first and second dictionaries as the left and right sub-trees.

```

trait BalancedDict[K, V, N] extends TreeDict[K, V, (K, V, N)]
  with JoinBasedDict[K, V, Tree[(K, V, N)], FocusHint[K, V, Tree[(K, V, N)]]] {
  def key(e: Entity): K = e._1
  def value(e: Entity): V = e._2
  def info(e: Entity): N = e._3
  def zeroInfo: N
  def info(e: D): N = e match {
    case Leaf() => zeroInfo
    case Bin(_, entity, _) => info(entity)
  }
  def newInfo(left: N, right: N): N
  def rotateLeft(tree: D): D = tree match {
    case Bin(l, e1, Bin(l2, e2, r2)) =>
      bin(bin(l, key(e1), value(e1), l2), key(e2), value(e2), r2)
    case _ => throw new Exception("Not left rotatable")
  }
  def rotateRight(tree: D): D = // elided for brevity
  def isHeavier(left: D, right: D): Boolean
  def isVeryHeavier(left: D, right: D): Boolean
  def joinRight(left: D, key: K, value: V, right: D): D =
    if(!isHeavier(left, right)) bin(left, key, value, right)
    else {
      val hint = middle(left)
      val (leftLeft, leftRight) = (before(hint), after(hint))
      val (k, v) = current(hint)
      val newRight = joinRight(leftRight, key, value, right)
      if(isHeavier(newRight, leftLeft)) {
        val newHint = middle(newRight)
        if(isVeryHeavier(before(newHint), after(newHint)))
          rotateLeft(bin(leftLeft, k, v, rotateRight(newRight)))
        else rotateLeft(bin(leftLeft, k, v, newRight))
      } else bin(leftLeft, k, v, newRight)
    }
  }
  def joinLeft(left: D, key: K, value: V, right: D): D = // elided for brevity
  def bin(left: D, key: K, value: V, right: D): D =
    Bin(left, (key, value, newInfo(info(left), info(right))), right)
  def join(left: D, key: K, value: V, right: D): D =
    if(isHeavier(left, right)) joinRight(left, key, value, right)
    else if (isHeavier(right, left)) joinLeft(left, key, value, right)
    else bin(left, key, value, right)
}

```

■ **Figure 22** The generalized interface for hinted dictionaries using balanced binary trees.

8.3 Balanced trees

Figure 22 shows the generalized implementation for balanced-tree-based hinted dictionaries. This interface subsumes AVL trees and WBB trees. It is possible to implement Red-Black tree and Treaps as hinted dictionaries by appropriately overriding the `join` method, however, as it was shown that AVLs and WBBs have superior performance in comparison with them [4], we only present their implementation in this paper.

The interface of `BalancedDict` accepts the extra parameter `N` for the extra information kept by the tree nodes. For example, AVL trees store the height of the tree in each node and WBB trees store the size of the tree as the weight information.

```

class AVLDict[K, V](implicit override val ord: Orderable[K], val mon: Monoid[V])
  extends BalancedDict[K, V, Int] {
  type N = Int
  def zeroInfo: Int = 0
  def newInfo(left: N, right: N): N = math.max(left, right) + 1
  def isHeavier(left: D, right: D): Boolean = info(left) > info(right) + 1
  def isVeryHeavier(left: D, right: D): Boolean = info(left) > info(right)
}
class WBBDict[K, V](implicit override val ord: Orderable[K], val mon: Monoid[V])
  extends BalancedDict[K, V, Int] {
  type N = Int
  val ALPHA = 0.29
  val RATIO = ALPHA / (1 - ALPHA)
  val BETA = (1 - 2 * ALPHA) / (1 - ALPHA)
  def zeroInfo: Int = 1
  def newInfo(left: N, right: N): N = left + right - 1
  def isHeavier(left: D, right: D): Boolean = RATIO * info(left) > info(right)
  def isVeryHeavier(left: D, right: D): Boolean =
    info(left) > BETA * newInfo(info(left), info(right))
}

```

■ **Figure 23** The implementation for hinted dictionaries based on AVL and WBB trees.

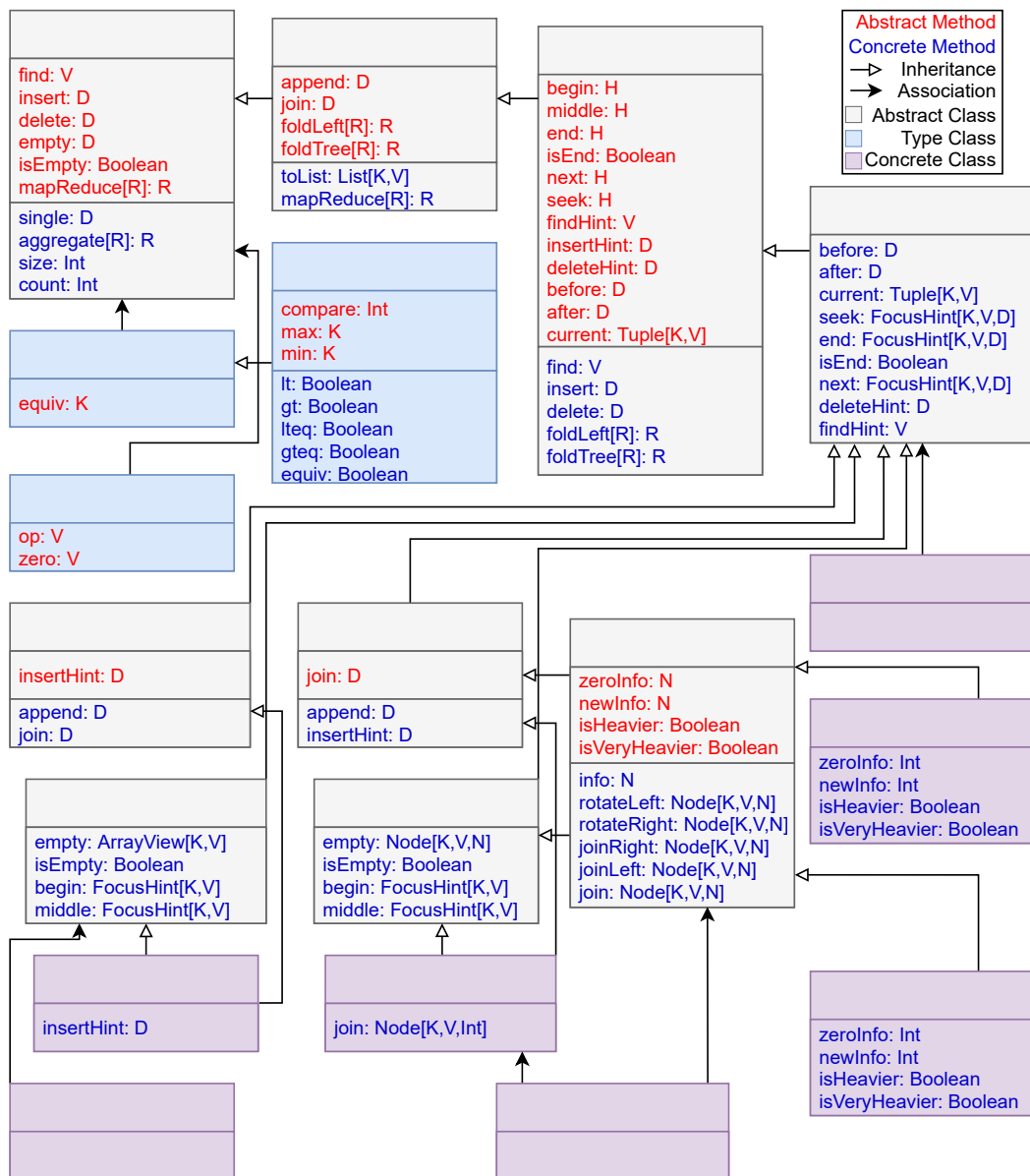
In order to preserve correct bookkeeping information, the smart constructor `bin` is added. This method invokes the abstract method `newInfo` in order to compute the updated information based on the information of the sub-trees. This method needs to be overridden by concrete balanced tree implementation choices.

The `join` method starts with checking if the tree is unbalanced towards either of its sub-trees. If this is not the case, the smart construct `bin` is invoked to simply construct the new root node. However, if either of the sides is heavier, an appropriate recursive method is invoked in order to take care of possible rotations.

In Figure 22 we show the implementation of `joinRight`, which is invoked when the left sub-tree is heavier than the right sub-tree. The implementation of `isHeavier` is again postponed to the concrete implementation of a balanced tree. The rest of the implementation of `joinRight` mirrors a generalized version of what has been reported before in [4]. An interesting case is when there needs to be a double rotation involved. This is checked by `isVeryHeavier`, which needs to be implemented by a concrete balanced tree implementation.

Figure 23 shows the implementation of AVL and WBB trees using the generalized interface mentioned above. The AVL tree maintains the height of the tree as the extra information. The height of a new tree is computed by incrementing the maximum height of its sub-trees by one. A sub-tree is heavier than another sub-tree when its height is more than an increment of the height of the other sub-tree. And finally, a node needs double rotation only if the height of its left sub-tree is more than the height of its right sub-tree.

The WBB tree considers the size of the tree (added by one) as the extra information, referred to as weight [1]. The updated weight is computed by adding the weight of sub-trees (decremented by one). The `RATIO` and `BETA` parameters control whether rotation or double rotation need to be performed. The values for these parameters specify the trade-off between the tree being in a perfect balance and the number of re-balancing invocations. There were follow up research on fixing the balancing issues related to the parameter values originally suggested by Adams [6, 16]. We use the parameters reported by [4].



■ **Figure 24** The bird's eye view of all the interfaces.

8.4 Putting It All Together

Finally, we give an overall picture of hinted dictionaries by connecting all the pieces together. Figure 24 shows the different interfaces defined throughout this paper. To reduce the number of classes, we merged the definition of several interfaces with each other (e.g., `DictIterations` is merged with `Dict`). Crucially, there is no cake-pattern-based interface in Figure 24, as this technique is not essential for implementing hinted dictionaries.


```

template<class K, class V, class Compare, class Mon>
class array_dict {
private:
    typedef std::vector<std::pair<K, V>> vector_t;
    typedef std::pair<int, bool> hint_t;
    int lower; int upper;
    vector_t buffer;
public:
    void insert(K& key, V& value);
    hint_t insert_hint(hint_t& hint_obj, K& key, V& value);
    bool is_end(hint_t& hint_obj);
    template<SearchMethod search_method>
    hint_t seek(K& key);
    void after(hint_t& hint_obj);
    template<class R, class Func>
    void inplace_fold_left(R& z, Func op);
    /* Elided for brevity */
}

```

■ **Figure 25** The interface of sorted-array-based dictionaries in C++.

9 Evaluation

In this section, we evaluate the performance of hinted dictionaries. First, we present an efficient C++ prototype for them. Then, we show the experimental results by comparing our C++ implementation with competitors for set-set and sparse vector operations.

9.1 Tuned Implementation in C++

The hinted dictionaries can have an efficient low-level implementation in C++. We provide the following hinted dictionary implementations: 1) `array_dict`, an array-based implementation that uses `std::vector<std::pair<K, V>>` as the underlying representation, and 2) `wbb_dict`, a tree-based implementation based on WBBs [9, 1].

We do not use the hierarchy presented in Figure 24 for performance reasons; we merge the definitions of all the parent interfaces of hinted dictionaries into `array_dict` and `wbb_dict`. The implementation for `wbb_dict` stays very similar to the one presented in Section 8.3. However, we have applied the following performance tuning tricks for `array_dict`, the interface of which is shown in Figure 25.

Pointer-Based Hints. In Section 7.1 we presented `FocusHint` objects that materialize the entire dictionaries before and after a hint object. However, this design results in unnecessarily copying of arrays (cf. Section 8.1). The `array_dict` implementation only maintains a pointer to the appropriate place by using pointer-based hint objects of type `std::pair<int, bool>`. If the hint object points to an actual element of the hinted dictionary, the first element of the pair specifies its index and the second element is set to `true`. For keys that do not exist in the dictionary (i.e., the associate value is the zero element of the underlying monoid), the first element is the index of an element with the *least upper bound* key and the second element is set to `false`.

Binary Search vs. Linear Search. In Section 7.2 we used binary search (by calling `middle`) in order to implement `seek`. However, as we observe in the next section, in many cases it could be beneficial to use linear search. We provide a template parameter for the `seek` method in order to specify the underlying search method.

In-Place Updates. Finally, we use in-place updates to improve the performance in the following cases. First, the methods that return a subset of the hinted dictionary (`before` and `after`) can perform an in-place modification of the boundary of the dictionary (`lower` and `upper`). Second, the aggregation-based methods that produce dictionaries can use a single mutable dictionary and modify it in-place, instead of passing around new dictionaries (cf. `inplace_fold_left` in Figure 25).

Constant-time Size. The `array_dict` implementation can compute the size of the dictionary by evaluating `upper - lower`. Similarly, `wbb_dict` can compute the size using the meta data (`info`). However, both dictionaries still require iterations for `count` (i.e., the number of elements with a non-zero value). Thanks to the fast size computation, we can make sure that we always iterate over smaller dictionaries for all binary operations over dictionaries (e.g., set-set operations).

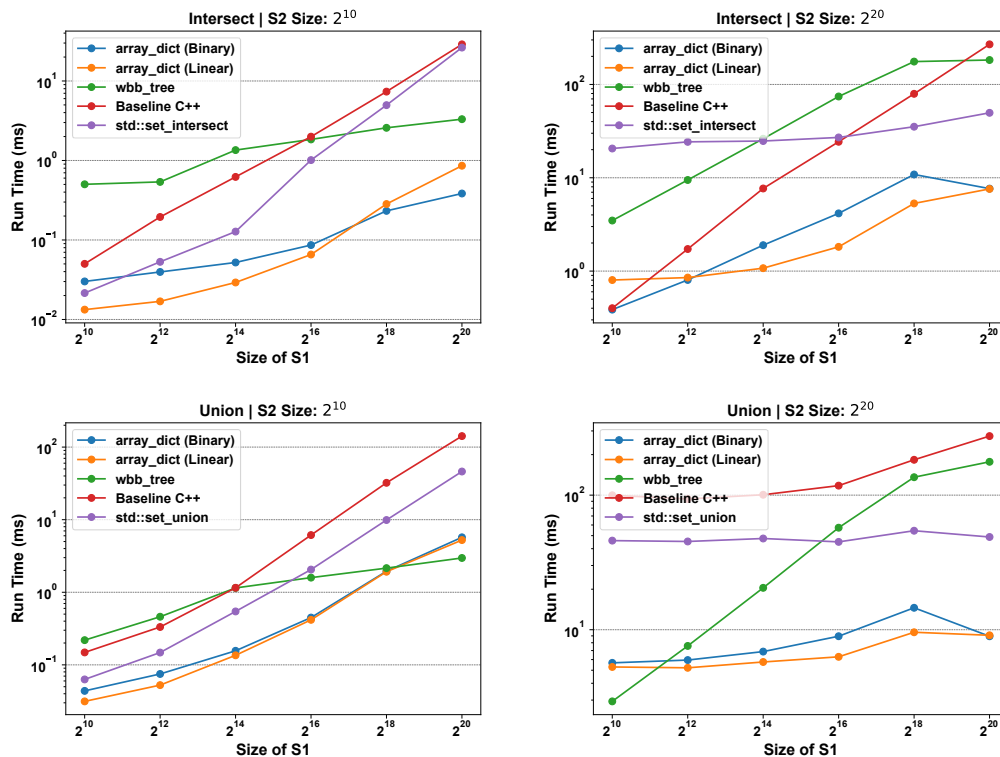
9.2 Experimental Results

Experimental Setup. We run our experiments on a machine running Ubuntu 20.04.3 equipped with an Intel Core i5 CPU running at 1.6GHz, 16GB of DDR4 RAM. We use G++ 9.4.0 for compiling the generated C++ code using the `O3` flag. We also use SciPy 1.8.0 (on Python 3.8.10) as the competitor.

Workloads. We consider the following set-set and sparse vector operations: 1) set-set union, 2) set-set intersection, 3) sparse vector addition, 4) sparse vector element-wise multiplication, and 5) sparse vector inner product. For all the experiments, we generate randomly synthetic data by varying the size of sets and the density of vectors. We run all the experiments for ten times and measure their average run time.

Competitors. We consider the following alternatives of our C++ prototype and other frameworks:

- **array_dict (Linear):** array-based dictionary with linear-search-based seek.
- **array_dict (Binary):** array-based dictionary with binary-search-based seek.
- **wbb_dict:** tree-based dictionary that uses a WBB-based representation.
- **Baseline C++:** a baseline implementation using the operations provided by `std::set` (for the set experiments) or `std::map` (for the sparse vector experiments).
- **std::set_intersect, std::set_union:** set-set operations provided by the standard library of C++. As input arguments we use `std::set` collections.
- **SciPy:** sparse linear algebra operators provided by the SciPy library. A sparse vector is represented as a row matrix with a CSR (Compressed Sparse Row) or a column matrix with a CSC (Compressed Sparse Column) format.



■ **Figure 26** Experimental results for set-set operations.

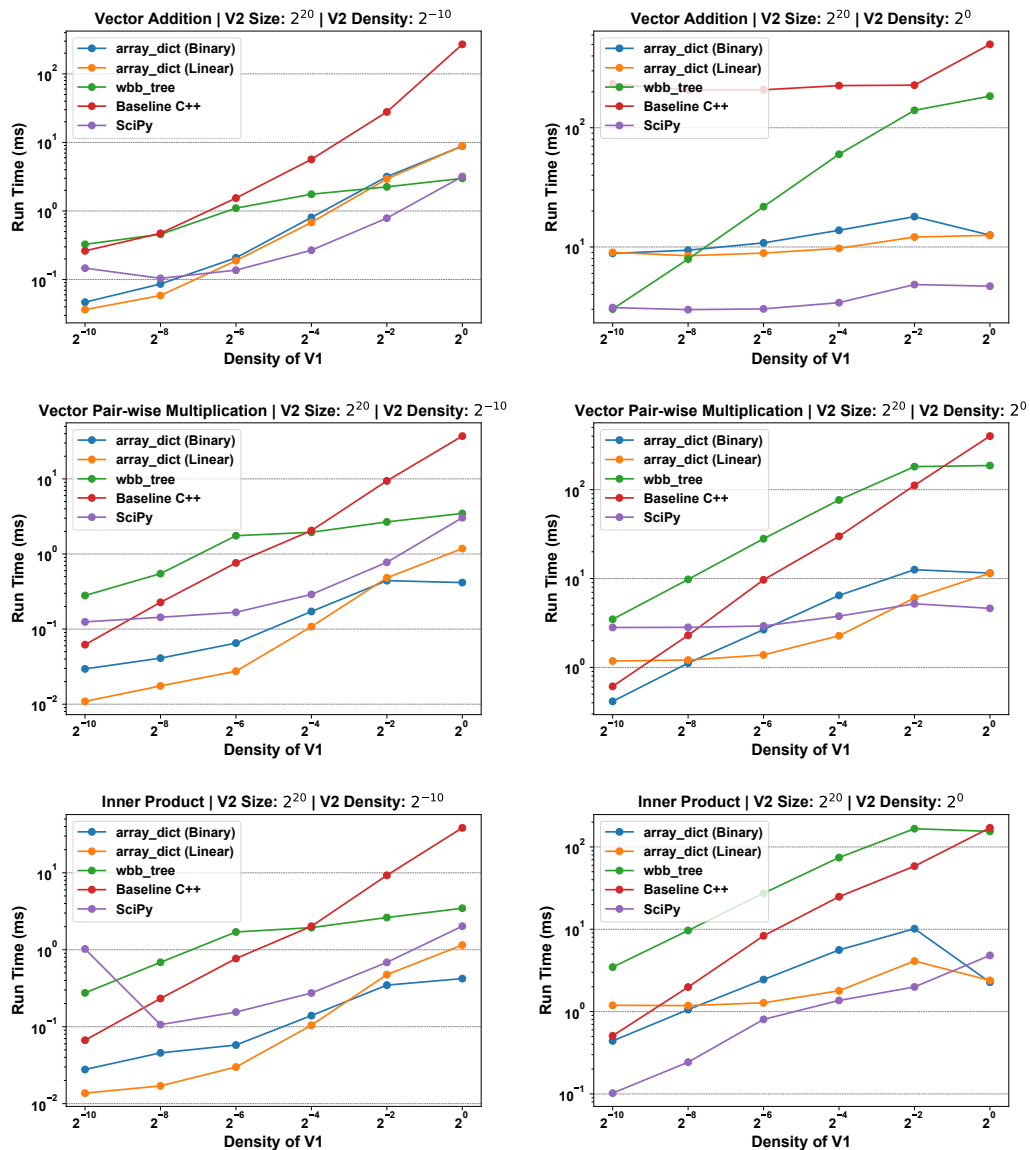
Set-Set Experiments. Figure 26 shows the results for the union and intersection operations over sets. We make the following observations. First, in most cases, we observe a superior performance for the `array_dict` implementations. In most cases, the linear-search-based approach has better performance. However, as the difference between the size of sets widens, the binary-search starts showing better performance. This is because of the additional run-time improvements caused by skipping irrelevant elements.

Furthermore, we observe that the `wbb_dict` implementation does not show superior performance in most cases. As shown before [4], one of the advantages of such join-based implementations is their amenability to parallelism that we leave for the future.

Finally, we observe an asymptotically improved performance over the baseline C++ implementation. This is thanks to turning $O(\log n)$ lookup and insertion operations into amortized constant-time ones. The implementations provided by the standard library of C++ suffer from a similar issue. In addition, due to a concurrent linear iteration over both sets, they show worse performance for the sets with a large difference in their sizes.

Sparse Vector Experiments. The implementations for sparse vector addition and element-wise multiplication are identical to the ones for set union and intersection, respectively. The sparse vector addition uses real number addition instead of boolean disjunction, and the element-wise multiplication uses real number multiplication instead of boolean conjunction. The results for vector inner product are also very similar to the ones for the element-wise multiplication (cf. Figure 27).

The SciPy framework uses a CSR format for representing all vectors, except for the second operand of the vector inner product. This is because the second vector needs to be transposed which makes the CSC format a better representation. Overall, this framework



■ **Figure 27** Experimental results for sparse vector operations.

shows superior performance for vectors with a large density. This can be related to their better storage layout (struct of array instead of array of struct) that leads to improved cache locality. The `array_dict` variants show better performance for vectors with a higher degree of sparsity.

10 Conclusion and Outlook

In this paper, we introduced hinted dictionaries, a unified technique for implementing ordered dictionaries. We have shown how hinted dictionaries unify the existing techniques from both imperative and functional languages. These dictionaries can be used as the collection type for data-intensive workloads. It would be interesting to see the usage of such data structures for real-world use-cases such as query processing (relying on relations in the form of sets and bags) as well as sparse linear algebra (relying on sparse vectors and matrices).

The performance improvement offered by hinted dictionaries does not come for free. The programmers must be careful on how to use hinted dictionaries. As presented in Section 4, certain preconditions need to be preserved for hint objects. Violating these preconditions by the programmers destroys the invariants of hinted dictionaries leading to runtime errors or, even worse, undefined behaviour, which can hinder the productivity of programmers. One interesting future direction is to statically detect the violation of the hinted dictionaries' invariants.

Furthermore, we envision the following future directions for hinted dictionaries. First, we plan to consider real-world applications that require batch processing of sets and maps, including relational query engines and sparse tensor processing frameworks. Furthermore, it would be interesting to use code generation and multi-stage programming techniques to generate low-level code. This way, one can automatically improve the performance by removing allocation of unnecessary intermediate objects (e.g., `FocusHint` objects) or to use in-place updates (cf. Section 9.1) from a purely functional implementation. Finally, for applications such as query processing the trade-offs between hashing and sorting have been debated for a long time. We believe hinted dictionaries provide a nice abstraction layer for DSLs based on dictionaries (e.g., the physical query plan of query engines) to automatically tune the choice of the underlying dictionary implementation [13].

References

- 1 Stephen Adams. Efficient sets – a balancing act. *JFP*, 3(4):553–561, 1993.
- 2 Georgy Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Proc. of the USSR Academy of Sciences*, 145:263–266, 1962. In Russian, English translation by Myron J. Ricci in *Soviet Doklady*, 3:1259-1263, 1962.
- 3 Rudolf Bayer. Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.
- 4 Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *SPAA '16*, pages 253–264, 2016.
- 5 Sébastien Doeraene. A type-parametric unboxed Option type for Scala. <https://github.com/sjrd/scala-unboxed-option>, 2019. Accessed: 2021-10-11.
- 6 Yoichi Hirai and Kazuhiko Yamamoto. Balancing weight-balanced trees. *Journal of Functional Programming*, 21(3):287–307, 2011.
- 7 Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.
- 8 Hećate Moonlight. unpacked sum types · Wiki · Glasgow Haskell Compiler / GHC. <https://gitlab.haskell.org/ghc/ghc/-/wikis/unpacked-sum-types>, 2021. Accessed: 2021-10-11.
- 9 Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973.
- 10 Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *ACM Sigplan Notices*, 45(10):341–360, 2010.
- 11 Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.
- 12 Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *PACMPL*, 6(OOPSLA1):1–33, 2022.
- 13 Amir Shaikhha, Marios Kelepeshis, and Mahdi Ghorbani. Fine-tuning data structures for query processing. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, CGO 2023*, pages 149–161, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3579990.3580016.

28:30 Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

- 14 Boost Software. Class template `flat_map` - 1.79.0. https://www.boost.org/doc/libs/1_79_0/doc/html/boost/container/flat_map.html, 2018. Accessed: 2021-10-5.
- 15 Milan Straka. The performance of the haskell containers package. *ACM Sigplan Notices*, 45(11):13–24, 2010.
- 16 Milan Straka. Adams’ trees revisited. In *International Symposium on Trends in Functional Programming*, pages 130–145. Springer, 2011.
- 17 Yihan Sun, Daniel Ferizovic, and Guy E Belloch. PAM: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 290–304, 2018.

Semantics for Noninterference with Interaction Trees

Lucas Silver ✉

University of Pennsylvania, Philadelphia, PA, USA


Paul He ✉ 

University of Pennsylvania, Philadelphia, PA, USA

Ethan Cecchetti ✉ 

University of Maryland, College Park, MD, USA

University of Wisconsin – Madison, WI, USA

Andrew K. Hirsch ✉ 

State University of New York at Buffalo, NY, USA

Steve Zdancewic ✉ 

University of Pennsylvania, Philadelphia, PA, USA

Abstract

Noninterference is the strong information-security property that a program does not leak secrets through publicly-visible behavior. In the presence of effects such as nontermination, state, and exceptions, reasoning about noninterference quickly becomes subtle. We advocate using *interaction trees* (*ITrees*) to provide compositional mechanized proofs of noninterference for multi-language, effectful, nonterminating programs, while retaining executability of the semantics. We develop important foundations for security analysis with *ITrees*: two *indistinguishability* relations, leading to two standard notions of noninterference with adversaries of different strength, along with metatheory libraries for reasoning about each. We demonstrate the utility of our results using a simple imperative language with embedded assembly, along with a compiler into that assembly language.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Security and privacy → Logic and verification; Security and privacy → Information flow control

Keywords and phrases verification, information-flow, denotational semantics, monads

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.29

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.6>

Software (Source Code): <https://github.com/DeepSpec/InteractionTrees/tree/secure>

archived at sw.h1.dir:5cdf25ba007aa5744f02131d87740ca72493488c

Funding This work was funded in part by the NSF under the award 1521539 (Weirich, Zdancewic, Pierce).

1 Introduction

Information-flow guarantees state that programs respect the information-security specifications of their inputs and outputs. The most basic is *noninterference*, which states that secret data cannot influence publicly observable behavior. There are many languages designed to enforce information-flow properties, guaranteeing that programs treat their sensitive inputs correctly [28, 39, 40]. The importance of information-security properties has increasingly led to verification efforts for such languages and systems [7, 20]. These efforts, however, are mostly limited to source-level guarantees for a single language. For security guarantees to be meaningful, the entire language toolchain must support them.



© Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 29; pp. 29:1–29:29



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



One of the key decisions when formalizing any effectful, possibly-nonterminating language is the choice of representation. Much prior work focuses on operational semantics defined as a relation on syntax, or on trace models defined as a predicate over lists or streams of observations [21, 25, 36]. However, such definitions often require auxiliary constructs, like program counters or evaluation contexts, making proofs brittle and hard to compose. These concerns are particularly pronounced for information-security properties, which often rely on subtle definitions with delicate correctness proofs. The complexity of multi-language settings further complicates the already-fraught choice of language representation.

Interaction Trees (ITrees) [57, 59] provide an alternative: a runnable denotational semantics for effectful, potentially-nonterminating programs, with a library implemented in Coq [29]. Intuitively, ITrees represent programs as interactions with the environment. At a technical level, ITrees are a coinductive data type based on free monads [50]. Programs are either done and provide a return value, emit an *event* to the environment and continue once the environment provides a response, or produce a “silent event,” allowing ITrees to represent (silently) diverging programs in strongly normalizing metalanguages. By interpreting the events into a suitable monad [31], ITrees can express the semantics of diverse programming-language features, and thus many different languages. This versatility makes ITrees well-suited to cross-language reasoning [57] and reasoning about real-world toolchains [59, 24].

ITrees come equipped with a notion of program equivalence based on *weak bisimilarity*, which considers programs equivalent if they differ only by a finite number of silent steps. Properties like noninterference, however, require more nuanced reasoning because some program behaviors are visible to an attacker while others are not.

This work introduces two *indistinguishability* relations for ITrees to capture these intuitions: one *progress-sensitive* and one *progress-insensitive*. These definitions – motivated by corresponding notions found in the information-flow security literature [56, 55, 45] – adapt the notion of bisimilarity to account for what information is available to an adversary. They require delicate treatment of the interplay between nontermination and the interactions of a program with its environment. Progress-sensitive noninterference is a very strong guarantee, but is overly restrictive for many real-world programming tasks. For instance, it generally disallows loops that depend on secret data. Progress-insensitive noninterference is less demanding, but provides considerably less security [6].

While the definitions of ITrees and our indistinguishability relations are coinductive, we provide metatheoretic results allowing a proof engineer to reason with these relations without manual coinduction. These results further connect these indistinguishability relations to the standard ITrees notion of bisimilarity, providing compatibility with existing results.

We validate this design with a simple toolchain for cross-language noninterference. The toolchain consists of a simple imperative language, IMP, and a simple assembly language, ASM. There are two type systems for IMP and a compiler from IMP to ASM. One type system enforces progress-sensitive noninterference and the other enforces progress-insensitive noninterference. In addition to standard information flow typing rules, the type systems allow for *semantic typing*: any semantically secure program can be considered well typed. This flexibility allows IMP to support embedded ASM blocks without giving a type system to ASM, and it demonstrates the powerful semantic composition of our security reasoning. We further verify that our IMP-to-ASM compiler preserves both kinds of noninterference. This preservation relies only on semantic security, not the type system, which is required to allow for security preservation with semantic typing.

To further demonstrate the utility of our approach, we include exceptions in IMP. Exceptions show how our indistinguishability semantics interact with effects that may alter control flow, which are a particular challenge for information-flow reasoning. This inclusion also requires an extension to the ITrees library that is orthogonal to the security extensions.

Section 2 reviews background on information-flow control and ITrees, the IMP language, and its semantics defined with ITrees. The contributions of this paper are as follows.

- Section 3 extends the ITrees library with exceptions and exception handlers.
- Section 4 adapts ITrees metatheory to reason about security guarantees, defining progress-sensitive and progress-insensitive notions of indistinguishability and noninterference.
- Section 5 uses ITrees and the new relations to prove the security of two standard information-flow type systems for IMP.
- Section 6 extends Xia et al.’s [57] simple compiler from IMP to ASM with exceptions and print effects. We then show that Xia et al.’s notion of compiler correctness immediately implies security preservation using only the metatheory of indistinguishability.

Finally, Section 7 discusses related work and Section 8 concludes. All definitions and theorems described in this paper have been formalized in Coq.

2 Background

We now review background on information-flow control, interaction trees, and IMP.

2.1 Information-Flow Control

We represent information-security policies using a set of *information-flow labels* \mathcal{L} that must form a preorder. That is, there is a reflexive, transitive relation \sqsubseteq (pronounced “flows to”) on labels where $\ell \sqsubseteq \ell'$ means that any *adversary* who can see information with label ℓ' can also see information with label ℓ . We also identify adversaries with labels. An adversary at label ℓ can only see information with labels that flow to ℓ . Information-flow systems use a variety of orderings, including simply “public” and “secret,” subsets of permissions [62], lattices over principals making up a system [33, 5, 49], and orderings based on logical implication [39].

The classic information-flow security policy is *noninterference*: if an adversary cannot distinguish a program’s inputs, they should not be able to distinguish its outputs or its interactions with the environment. Because information-flow labels determine which data an adversary can observe, a semantic version of noninterference requires a semantic model of information-flow labels. Sabelfeld and Sands [46] suggest modeling labels as partial equivalence relations (PERs) on terms. PERs are relations that are symmetric and transitive, but not necessarily reflexive. PERs act like equivalence relations on a subset of their domain. For information-flow security, such PERs are called “indistinguishability relations.”

This model further asserts that indistinguishable programs take indistinguishable inputs to indistinguishable outputs. That is, related programs, applied to related inputs, produce related computations. This closure property allows a semantic version of noninterference to be defined as self-relation of a program. A program is related to itself – and noninterfering – if and only if, for every adversary, given any two inputs an adversary cannot distinguish, it produces two computations that adversary cannot distinguish.

As we will see in Section 4, indistinguishability gives a natural way to reason about noninterference using ITrees. Requiring every indistinguishability relation to be a PER, however, corresponds to strong assumptions about the adversary. In particular, it requires that the adversary be able to distinguish a program that silently diverges from a program that takes arbitrarily long to produce an observable interaction with its environment. Noninterference against this strong adversary is known as *progress-sensitive* noninterference. While this strength provides more security, enforcing progress-sensitive noninterference results in a prohibitively expensive programming model [45, 55, Section 5.1]. To allow for enforcement of *progress-insensitive* noninterference, the indistinguishability model is often relaxed to not require transitivity [54, 42, 16].

2.2 Basic Definitions for Interaction Trees

Interaction Trees (ITrees) [57] are a coinductive data structure designed to give denotational semantics to effectful, possibly divergent programs. ITrees model such computations as branching trees where internal nodes represent *events*, or interactions with the environment, with a branch for each different possible response from the environment. The use of coinduction means that these trees can be infinite, modeling diverging programs. Because ITrees give a denotational semantics to programs, they are a language-agnostic view of programs. Thus, we can use ITrees as a common domain for multiple languages, allowing us to reason about how those languages interact.

The type of an ITree includes an event signature E and a result type R . The result type simply specifies the output type if the program halts normally. The event signature E defines the interface by which the environment interacts with the program. $E : Type \rightarrow Type$ is a type transformer that takes an answer type A and returns $E A$, the type of an event that produces a value of type A . For example, the event signature, **stateE**, modeling a state effect might have two constructors: **get** and **set**. A **get** event represents a state access that returns a number, so it has type **stateE**(\mathbb{N}). A **set** event represents an assignment that need not return any useful information, so it has type **stateE**(**unit**).

ITrees have the following constructors.

$$\frac{r : R}{\mathbf{ret} \ r : \mathbf{itree} \ E \ R} \quad \frac{t : \mathbf{itree} \ E \ R}{\tau \cdot t : \mathbf{itree} \ E \ R} \quad \frac{e : E \ A \quad k : A \rightarrow \mathbf{itree} \ E \ R}{\mathbf{Vis} \ e \ k : \mathbf{itree} \ E \ R}$$

In this paper, a double line in an inference rule means that it should be interpreted coinductively, while a single line is interpreted inductively, as usual. This definition, then, is a fully coinductive definition, since the only single-line definition is a base case.

The ITree **ret** r represents a program terminating with a value r . The ITree $\tau \cdot t$ represents a silent internal step of computation, followed by the ITree t . Because ITrees are a *coinductive* data structure, we can chain an infinite number of τ 's together in the ITree $t_{\text{spin}} = \tau \cdot t_{\text{spin}}$. Here, t_{spin} models a divergent program that causes no side effects. Finally, the ITree **Vis** $e \ k$ represents a visible event e of type $E \ A$ for some answer type A , followed by a continuation k that takes an answer of type A and produces an **itree** $E \ R$. Intuitively, k defines how the computation proceeds after the environment handles event e . Since k 's behavior may differ depending on the value returned by e , there is one possible computational “branch” for each value of type A . In this view, ITrees are potentially infinitely long trees.

For any event signature E , **itree** E forms a monad [31]. The unit operation is provided by the **ret** constructor, and the bind operation, written $m \gg= k$, is defined as a corecursive function which replaces every **ret** r in m with $k \ r$. We will also use the common monad notation $x \leftarrow t_1 ; t_2$ in place of $t_1 \gg= \lambda x.t_2$. ITrees satisfy the monad laws up to strong bisimulation, which we use as an equivalence on ITrees since they are potentially infinite objects. Two ITrees are strongly bisimilar when they have exactly the same shape (including the values returned at corresponding leaves).

In combination with the monad operations, another useful operation is **trigger**, which lifts an event into an ITree that immediately returns the environment's response:

$$\mathbf{trigger} \ e = \mathbf{Vis} \ e \ \mathbf{ret}$$

ITrees also support an *iteration* operation:

$$\mathbf{iter} : \forall A, B. (A \rightarrow \mathbf{itree} \ E \ (A \oplus B)) \rightarrow A \rightarrow \mathbf{itree} \ E \ B$$

Intuitively, **iter** *body* a acts as a do-while loop, running *body* on input a and either continuing with a new value of type A , or stopping with a final value of type B .

Expressions	$e ::= x \mid n \mid e + e \mid e - e \mid e * e$
Commands	$c ::= \text{skip} \mid x := e \mid c_1 ; c_2 \mid \text{while } (e) \text{ do } \{c\}$ $\quad \mid \text{if } (e) \text{ then } \{c_1\} \text{ else } \{c_2\} \mid \text{print}(\ell, e) \mid \text{inline } \{a\}$
Inlined Assembly	$a ::=$ (see Section 6)

■ **Figure 1** IMP syntax, where x is a variable, n is a number, and ℓ is an information-flow label.

2.3 Semantics for Imp with Security Labels

To explore how ITrees can help us verify noninterference properties, we will use a simple imperative language, IMP, as a running example and case study. Conveniently, previous work on both ITrees [57] and noninterference [45] use IMP as case studies, ensuring that the connection we make corresponds with existing tools and techniques in both domains. Our version of IMP, presented in Figure 1, includes features not present in the works cited above: the ability to print expressions to one of several output streams, and the ability to inline code from a simple assembly language. Section 3 will further extend IMP to allow throwing and catching exceptions. The output streams are indexed by information-flow labels, and we think of stream ℓ as being visible to any adversary at or above ℓ , but no others. Thus, printing secret information to a public stream leaks data.

The assembly language, ASM, is a simplification of standard assembly language. We allow an infinite number of registers, and we assume that the heap is addressed by variables, as in IMP. We also do not allow dynamic jumps, only jumps to fixed addresses. Beyond those simplifications, we include features similar to those in IMP: we allow printing to streams indexed by information-flow labels and, as we show later, the ASM semantics can model *uncaught* exceptions, both features necessary for correct compilation of IMP code. We discuss the syntax and semantics of ASM in more detail in Section 6.

As in languages like C, embedding ASM in IMP allows developers more control over the performance of their code. For instance, the simple compiler in Section 6 would compile the IMP program $y := x + 1 ; z := x + 2$ to an ASM program that loads data from x into a register twice, once for each assignment. Since Loads are relatively expensive, when the IMP code above appears in a critical loop a developer might replace it with the following ASM code:

```

START : LOAD   $0 ← x
        ADD    $0 ← $0, 1
        STORE  y ← $0
        ADD    $0 ← $0, 1
        STORE  z ← $0
        JMP    EXIT

```

This program starts from the START label, and terminates the program by jumping to the EXIT label. Unlike our compiler’s output, this custom ASM only has one load instruction.

Giving semantics to IMP using ITrees requires defining events representing possible interactions between an IMP program and its environment. IMP has three types of events: **stateE** for the heap state, **regE** for the register state, and **printE** for output. There are two constructors for **stateE** events, one for reading and one for writing.

$$\text{get} : \text{var} \rightarrow \text{stateE}(\mathbb{N}) \qquad \text{set} : \text{var} \rightarrow \mathbb{N} \rightarrow \text{stateE}(\text{unit})$$

The **regE** events require another two constructors, again one for reading and one for writing.

$$\text{getreg} : \text{reg} \rightarrow \text{regE}(\mathbb{N}) \qquad \text{setreg} : \text{reg} \rightarrow \mathbb{N} \rightarrow \text{regE}(\text{unit})$$

$$\begin{array}{l}
\boxed{\llbracket e \rrbracket_e : \text{itree progE } \mathbb{N}} \qquad \boxed{\llbracket c \rrbracket_c : \text{itree progE unit}} \\
\\
\llbracket x \rrbracket_e = \text{trigger get}(x) \qquad \llbracket \text{skip} \rrbracket_c = \text{ret } () \\
\llbracket n \rrbracket_e = \text{ret } n \qquad \llbracket x := e \rrbracket_c = n \leftarrow \llbracket e \rrbracket_e ; \text{trigger set}(x, n) \\
\llbracket e_1 + e_2 \rrbracket_e = x \leftarrow \llbracket e_1 \rrbracket_e ; \\
\qquad y \leftarrow \llbracket e_2 \rrbracket_e ; \\
\qquad \text{ret } (x + y) \qquad \llbracket \text{print}(\ell, e) \rrbracket_c = n \leftarrow \llbracket e \rrbracket_e ; \text{trigger print}(\ell, n) \\
\llbracket c_1 ; c_2 \rrbracket_c = \llbracket c_1 \rrbracket_c ; \llbracket c_2 \rrbracket_c \\
\llbracket \text{if } e \\
\text{then } \{c_1\} \\
\text{else } \{c_2\} \rrbracket_c = \begin{array}{l} \text{if } n \neq 0 \\ n \leftarrow \llbracket e \rrbracket_e ; \text{then } \llbracket c_1 \rrbracket_c \\ \text{else } \llbracket c_2 \rrbracket_c \end{array} \\
\llbracket \text{while } (e) \text{ do } \{c\} \rrbracket_c = \text{iter} \left(\begin{array}{l} \lambda_. n \leftarrow \llbracket e \rrbracket_e ; \\ \text{if } n \neq 0 \\ \text{then } (\llbracket c \rrbracket_c ; \text{ret inl}()) \\ \text{else ret inr}() \end{array} \right) () \\
\llbracket \text{inline } \{a\} \rrbracket_c = \llbracket a \rrbracket_{\text{asm}}
\end{array}$$

■ **Figure 2** Imp denotational semantics.

There is only one constructor for `printE` events: `print` : $\mathcal{L} \rightarrow \mathbb{N} \rightarrow \text{printE}(\text{unit})$.

As IMP programs can produce all three types of events, we combine them with disjoint union. The resulting event type for IMP programs is `progE` = `regE` \oplus `stateE` \oplus `printE`. For notational simplicity, we elide the injection operator when using these compound events.

Figure 2 presents the denotation of IMP using these events. Note that there are two denotation functions: $\llbracket \cdot \rrbracket_e$ for expression and $\llbracket \cdot \rrbracket_c$ for commands. As expressions produce numbers and commands have no output, $\llbracket \cdot \rrbracket_e$ produces computations of type `itree progE` \mathbb{N} , while $\llbracket \cdot \rrbracket_c$ produces computations of type `itree progE` `unit`. The function $\llbracket \cdot \rrbracket_{\text{asm}}$ gives ITree-based semantics to ASM. Its full definition can be found in the work of Xia et al. [57]; we discuss the modifications necessary to accommodate our changes in Section 6.

The denotation for expressions is fairly straightforward, and, importantly for proofs, completely compositional – an expression’s meaning is constructed from that of its subexpressions. The denotation of a variable is a `get` event, a literal n becomes `ret` n , and arithmetic expressions simply denote each argument and return the resulting value using `bind`.

Most commands are equally simple and compositional. `skip` is an immediate `ret`. Both assignment and `print` first denote the argument and then bind the result into the appropriate event. Sequencing is implemented with `bind` on a unit value that we elide. Conditionals first denote the condition, and then return the denotation of either the left or right command depending on the result.

Loops are more complex and make use of the `iter` combinator. The combinator expects a function that returns `itree progE` (`unit` \oplus `unit`), where a left value indicates “continue” and a right value indicates that the loop should terminate. The function given to `iter` first computes the value of the loop’s guard expression. If the value is not zero, it sequences a single denotation of the loop body with `ret inl`(), indicating the loop should continue. Otherwise, if the value is zero, it signals to halt the iteration with `ret inr`() .

2.4 Handlers and Interpretations

The events in an ITree can be thought of as a kind of syntax. Even though we give them names that suggest certain behaviors, like `get` and `set`, nothing about their structure enforces this behavior. Consider the ITree `trigger set(x, 0) ; trigger get(x)`: while the names suggest

that the result of this `get` should be 0, it actually produces a tree with one branch for every natural number. Likewise, the $\text{ITree } \llbracket c \rrbracket_c$ representing an IMP program c does not fully express the behavior we would expect from c because it has uninterpreted state events.

The behavior of events is determined by a function called an *event handler* from events to effectful computations. As is standard, we represent effectful computations as elements of a monad M , giving an event handler the type $\forall A. E A \rightarrow M A$. For example, consider h_{prog} which uses the standard monadic interpretation of state to interpret `progE` events:

$$\begin{aligned} h_{prog}(\text{get}(x)) &= \lambda(r, h). \text{ret } (r, h, h(x)) \\ h_{prog}(\text{set}(x, n)) &= \lambda(r, h). \text{ret } (r, h[x \mapsto n], ()) \\ h_{prog}(\text{getreg}(x)) &= \lambda(r, h). \text{ret } (r, h, r(x)) \\ h_{prog}(\text{setreg}(x, n)) &= \lambda(r, h). \text{ret } (r[x \mapsto n], h, ()) \\ h_{prog}(\text{print}(\ell, n)) &= \lambda(r, h). \text{trigger print}(\ell, n) ; \text{ret } (r, h, ()) \end{aligned}$$

Any event handler can be lifted to a function from ITrees to effectful computations using the `interp` function, which traverses an ITree , replacing each event with the effectful computation assigned by the handler. The full semantics of an IMP program is the *interpreted* ITree , `interp` $h_{prog} \llbracket c \rrbracket_c$.

2.5 Inlined Asm and Undefined Behavior

Adding support for inlined ASM code introduces a new complication to the semantics of IMP: undefined behavior. To analyze the correctness and security of a language toolchain, we need to define the behavior of source-level programs. The semantics defined in Section 2.3 and Section 2.4 do that for IMP as long as any inlined ASM has well-defined behavior. However, consider the following IMP program, which contains inlined ASM.

```
p = c ; inline { START : BRZ   $0 A1 A2
                  A1 : LOAD  X ← 0
                      JMP   EXIT
                  A2 : LOAD  X ← 1
                      JMP   EXIT }
```

The inlined ASM program looks at the value in register 0 and, if it is zero, jumps to address A1; otherwise it jumps to address A2. Thus, the value of X after executing program p depends on the value of register $\$0$ after c is executed. However, it is not clear what the register's value will be when this program is compiled and run, since reasonable compilers could use the register $\$0$ in different ways – or not at all – to compile the IMP command c , resulting in different register states. We thus consider inlining any ASM program that relies on the initial values of registers to be undefined behavior. We formalize this property in Section 5.3. We further take the same approach as `CompCert`,¹ and only verify the correctness and security of programs that are well-defined.

2.6 Weak Bisimulation

Much of the power of ITrees comes from their equational theory. While it is natural to reason about coinductive structures like ITrees using *bisimulation*, the “obvious” bisimulation relation is too strong for our needs. For example, the more complex operations we have

¹ Personal Communication with Xavier Leroy.

$$\begin{array}{c}
\frac{\mathcal{R}(r_1, r_2)}{E \vdash \text{ret } r_1 \approx_{\mathcal{R}} \text{ret } r_2} \\
\\
\frac{E \vdash t_1 \approx_{\mathcal{R}} t_2}{E \vdash \tau \cdot t_1 \approx_{\mathcal{R}} \tau \cdot t_2} \\
\\
\frac{e : E \ A \quad \forall(a : A), E \vdash k_1(a) \approx_{\mathcal{R}} k_2(a)}{E \vdash \text{Vis } e \ k_1 \approx_{\mathcal{R}} \text{Vis } e \ k_2} \\
\\
\frac{E \vdash t_1 \approx_{\mathcal{R}} t_2}{E \vdash \tau \cdot t_1 \approx_{\mathcal{R}} t_2} \\
\\
\frac{E \vdash t_1 \approx_{\mathcal{R}} t_2}{E \vdash t_1 \approx_{\mathcal{R}} \tau \cdot t_2}
\end{array}$$

■ **Figure 3** Inference rules for weak bisimulation.

introduced, like `iter` and `interp`, insert some (finite number of) silent internal τ steps, which would be convenient to ignore. For this reason, we often prefer to work with a coarser equivalence called *weak bisimulation*, or *equivalence-up-to-tau* (`eutt`), which ignores finite numbers of τ s when comparing two ITrees.

Weak bisimulation is defined by the inference rules in Figure 3, where the relation is parameterized by a relation \mathcal{R} used to compare return values. Furthermore, the event signature of the two ITrees is made explicit by the E parameter. The first three inference rules correspond to the three constructors of an ITree and are exactly the definition of strong bisimulation. The last two rules allow us to ignore any finite number of τ s. The fact that these rules are inductive rather than coinductive is crucial. If these rules were coinductive, we could use them to show that a diverging ITree with only τ constructors is equivalent to any other ITree. Using this technique of mixed induction and coinduction, coinductive rules may be used infinitely often, while inductive rules can only be used a finite number of times before either terminating with a base case or applying a coinductive rule.

Xia et al. [57] formalize the ITrees data structure and its metatheory in a Coq library,² providing a rich equational theory up to this definition of weak bisimulation. This theory allows users to prove termination-sensitive properties about ITrees without explicitly performing coinductive proofs, greatly reducing the proof burden.

3 Exceptions with Interaction Trees

As mentioned in Section 1, we include exceptions in IMP since they are an important example of an effect which can change the control flow. In this section, we show how to model exceptions with ITrees by adding `throw` and `catch` constructs to IMP as follows:

Commands $c ::= \dots \mid \text{throw}(\ell) \mid \text{try } \{c_1\} \text{ catch } \{c_2\}$

Note that the `throw` command includes an information flow label, specifying who may see the exception.

3.1 Exceptions as Halting Events

We model exceptions in ITrees as *halting events*. Recall from Section 2.2 that events create one branch for every possible response from the system. If an event has an uninhabited response type, then that continuation can never be run since the answer type has no values. We call such events *halting* because they force the computation to stop. We formalize this with the following lemma:

² This Coq development, as well as our extension of it, defines coinductive relations using the `paco` library [18, 60] for coinductive reasoning.

► **Lemma 1.** *Suppose A is an uninhabited type and e is an event of type E A , then given any continuations k_1 and k_2 and any return relation \mathcal{R} , $E \vdash \text{Vis } e \ k_1 \approx_{\mathcal{R}} \text{Vis } e \ k_2$.*

The continuation of a halting event cannot be run and has no effect on the computational content of the ITree. This allows a programmer to assign such an ITree any desired return type without changing its computational content. This property makes halting events useful for modeling (uncaught) exceptions: an exception can have any type and causes computation to stop. To represent exceptions using this strategy, we use an event type excE with only a single constructor $\text{exc} : \text{Err} \rightarrow \text{excE}(\emptyset)$ which takes the exception’s data payload and produces an event with an empty answer type. This allows us to define $\llbracket \text{throw}(\ell) \rrbracket_c = \text{trigger } \text{exc}(\ell)$.

3.2 Catching Exceptions

Real-world languages do not just throw exceptions, they also *handle* them. To implement exception handling in ITrees, we use a common monadic interpretation of exceptions: we allow programs to return either a standard return value or an exception. Specifically, we move from an ITree of type $\text{itree } (\text{excE } \text{Err} \oplus E) R$ to one of type $\text{itree } (\text{excE } \text{Err} \oplus E) (\text{Err} \oplus R)$ using interp to lift the following h_{exc} event handler to the entire ITree, as described in Section 2.4.

$$\begin{aligned} h_{\text{exc}} &: \forall A, (\text{excE } \text{Err} \oplus E) A \rightarrow \text{itree } (\text{excE } \text{Err} \oplus E) (\text{Err} \oplus A) \\ h_{\text{exc}}(\text{inl}(\text{exc}(e))) &:= \text{ret } \text{inl}(e) \\ h_{\text{exc}}(\text{inr}(e)) &:= x \leftarrow \text{trigger } \text{inr}(e); \text{ret } \text{inr}(x) \end{aligned}$$

Even though the resulting ITree cannot have exception events, we still assign it a type that allows them so it can cleanly compose with ITrees that do contain exception events. This choice allows monadic bind to apply exception handlers – which may themselves contain exception events – to any left values (exceptions) while leaving right values (normal returns) unmodified. The result is the following exception-handling combinator, where $\text{case } k_1 \ k_2$ chooses the continuation k_1 or k_2 if the return value is inl or inr , respectively.

$$\text{trycatch}(t, k_c) := \text{interp } h_{\text{exc}} \ t \gg= \text{case } k_c \ \text{ret}$$

This trycatch combinator has a straightforward metatheory. In particular, we show how it interacts with the constructors of ITrees, allowing proof engineers to reason about trycatch without using manual coinduction.

► **Theorem 2.** *The trycatch operator satisfies the following equivalences:*

$$\begin{aligned} E \vdash \text{trycatch}(\text{ret } r, k_c) &\approx= \text{ret } r \\ E \vdash \text{trycatch}(\tau \cdot t, k_c) &\approx= \text{trycatch}(t, k_c) \\ E \vdash \text{trycatch}(\text{Vis } \text{inr}(a) \ k, k_c) &\approx= \text{Vis } \text{inr}(a) \ \lambda x. \text{trycatch}(k(x), k_c) \\ E \vdash \text{trycatch}(\text{Vis } \text{inl}(\text{exc}(\varepsilon)) \ k, k_c) &\approx= k_c(\varepsilon) \end{aligned}$$

Finally, the trycatch operator provides a simple denotation of IMP’s try-catch blocks:

$$\llbracket \text{try } \{c_1\} \ \text{catch } \{c_2\} \rrbracket_c = \text{trycatch}(\llbracket c_1 \rrbracket_c, \lambda _ . \llbracket c_2 \rrbracket_c)$$

4 Indistinguishability of Interaction Trees

To leverage the common semantic domain of ITrees to guarantee the security of a toolchain, we define our indistinguishability relation purely semantically. Intuitively, for programs to be indistinguishable, they must return indistinguishable results and have indistinguishable interactions with their environments.

Since return values can be arbitrary types, we follow `eutt` by parameterizing indistinguishability over a *return relation* \mathcal{R} . For indistinguishability, \mathcal{R} describes when two values *appear* to be the same to the adversary. For example, consider a program that outputs a pair (a, b) where a is visible to Alice and b is visible to Bob, but not vice versa. The values $(1, 1)$ and $(1, 2)$ are not equal, but they are indistinguishable from Alice’s perspective, as she can only see the first element. We can represent Alice’s view of the output with a relation $\mathcal{R}_{\text{Alice}}$ defined by $\mathcal{R}_{\text{Alice}}((a, b), (a', b')) \iff a = a'$.

We could simply use `eutt` with a return relation \mathcal{R} modeling indistinguishability. The resulting relation would model an adversary who can only see some part of the program’s output, but it would require the two programs to interact with the environment in precisely the same way. Most settings, however, allow adversaries to see some interactions, but not others. For example, memory may be partitioned into a protected heap the adversary can never see, and an unprotected heap that it can see at all times. Reasoning about security when some events are visible and others are not requires changing `eutt` to account for what the adversary can observe.

4.1 Secure Equivalence Up-To Taus

Our indistinguishability relation is called *secure equivalence up-to tau* or `seutt`. In addition to a return relation, `seutt` is also parameterized by a label ℓ , representing what the adversary can see, and a *sensitivity function* ρ that maps events to labels, representing who may observe which events. Intuitively, two ITrees are related by `seutt` if the environment interactions appear the same to an adversary who can see events only at or below label ℓ , and the return values are related by \mathcal{R} . We write the relation as $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}}^{\ell} t_2$.

Notably, we base the relation on `eutt`, which makes it progress sensitive. Recall from Section 2.1 that progress-sensitive noninterference allows any adversary to determine if a program silently diverges, and is often prohibitively expensive to enforce. We will also define `pi-seutt`, a progress-insensitive version of `seutt`, in Section 4.3. The judgments take the same form, so we annotate the turnstile with a subscript ps or pi to distinguish them visually.

For presentation, we separate the rules for `seutt` into three groups: rules covering returns, τ s, and public events (Figure 4), rules covering secret events that do not halt the program (Figure 5), and rules covering secret halting events (Figure 6).

Public Events and Returns. When an adversary is able to see an event, indistinguishability acts just like weak bisimulation. The rules, found in Figure 4, are almost identical to the rules of `eutt`, but with the added requirement that any visible event be visible to the adversary. That is, we require $\rho(e) \sqsubseteq \ell$ in `PUBVIS`.

It might seem mysterious that we *require* the event to be visible in `PUBVIS`. But allowing this rule to apply no matter the visibility would allow the adversary too much power, since they would know that the same result is returned on both sides of the equivalence. As we will see, the rule for invisible events is stricter. We will also see how this strictness, when proving a program p indistinguishable from itself, corresponds to proving that the behavior

$$\begin{array}{c}
\text{[RET]} \frac{\mathcal{R}(r_1, r_2)}{E; \rho \vdash_{ps} \mathbf{ret} r_1 \approx_{\mathcal{R}}^{\ell} \mathbf{ret} r_2} \qquad \text{[TAUTAU]} \frac{E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}}^{\ell} t_2}{E; \rho \vdash_{ps} \tau \cdot t_1 \approx_{\mathcal{R}}^{\ell} \tau \cdot t_2} \\
\text{[PUBVIS]} \frac{\forall a, E; \rho \vdash_{ps} k_1(a) \approx_{\mathcal{R}}^{\ell} k_2(a) \quad e : E A \quad \rho(e) \sqsubseteq \ell}{E; \rho \vdash_{ps} \mathbf{vis} e k_1 \approx_{\mathcal{R}}^{\ell} \mathbf{vis} e k_2} \qquad \text{[TAUL]} \frac{E; \rho \vdash_{ps} \tau \cdot t_1 \approx_{\mathcal{R}}^{\ell} t_2}{E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}}^{\ell} t_2} \\
\text{[TAUR]} \frac{E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}}^{\ell} \tau \cdot t_2}{E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}}^{\ell} t_2}
\end{array}$$

■ **Figure 4** Inference rules for indistinguishability, where all events are visible.

$$\begin{array}{c}
\text{[PRIVVISTAU]} \frac{\forall a, E; \rho \vdash_{ps} k(a) \approx_{\mathcal{R}}^{\ell} t \quad e : E A \quad \neg \mathbf{empty}(A) \quad \rho(e) \not\sqsubseteq \ell}{E; \rho \vdash_{ps} \mathbf{vis} e k \approx_{\mathcal{R}}^{\ell} \tau \cdot t} \qquad \text{[PRIVVISINDL]} \frac{\forall a, E; \rho \vdash_{ps} k(a) \approx_{\mathcal{R}}^{\ell} t \quad e : E A \quad \neg \mathbf{empty}(A) \quad \rho(e) \not\sqsubseteq \ell}{E; \rho \vdash_{ps} \mathbf{vis} e k \approx_{\mathcal{R}}^{\ell} t} \\
\text{[PRIVVISVIS]} \frac{\forall (a:A)(b:B), E; \rho \vdash_{ps} k_1(a) \approx_{\mathcal{R}}^{\ell} k_2(b) \quad e_1 : E A \quad e_2 : E B \quad \rho(e_1) \not\sqsubseteq \ell \quad \rho(e_2) \not\sqsubseteq \ell \quad \neg \mathbf{empty}(A) \quad \neg \mathbf{empty}(B)}{E; \rho \vdash_{ps} \mathbf{vis} e_1 k_1 \approx_{\mathcal{R}}^{\ell} \mathbf{vis} e_2 k_2}
\end{array}$$

■ **Figure 5** Inference rules for indistinguishability, where events are not visible but answer types are inhabited.

of p does not differ in runs in *low-equivalent* environments. If we were to allow high events in PUBVIS, this would allow our proof to only consider the behavior of p in one environment, breaking our correspondence with information-flow security.

Private Events With Responses. When the adversary is *unable* to view an event, \mathbf{seutt} cannot act like \mathbf{eutt} . In this case, the rules are designed to formalize two intuitions. If the computation continues after a secret event, we should treat the event like a τ , since the adversary cannot observe either. If the event halts the computation, the event should be equivalent to a silently nonterminating computation.

The rules in Figure 5, along with symmetric analogues of PRIVVISTAU and PRIVVISINDL, handle the case where the event allows computation to continue – that is, the event’s answer type is inhabited. The first rule, PRIVVISTAU, relates a private event $\mathbf{vis} e k$ with a $\tau \cdot t$. In addition to requiring the event to be secret ($\rho(e) \not\sqsubseteq \ell$) and have a non-empty answer type ($\neg \mathbf{empty}(A)$), it also requires the continuation k produce an ITree indistinguishable from t for *every* possible response. This requirement ensures that the adversary’s future observations cannot depend on the response to the private event. Note that the requirement that A be non-empty does more than just specify when the rule applies. Without it, a private halting event would trivially satisfy this condition, allowing it to relate to any ITree with a τ in front. Since the adversary can determine when a program has halted, they should be able to distinguish, for example, a program that throws a private exception from a program which, after a τ , prints to a public channel. This rule ensures that this intuition holds.

$$\begin{array}{c}
E; \rho \vdash_{ps} \mathbf{Vis} \ e \ k \approx_{\mathcal{R}}^{\ell} t \\
e : E \ A \quad \mathit{empty}(A) \\
\rho(e) \not\sqsubseteq \ell \\
\hline
[\text{EMPVISTAU}] \frac{}{E; \rho \vdash_{ps} \mathbf{Vis} \ e \ k \approx_{\mathcal{R}}^{\ell} \tau \cdot t}
\end{array}
\qquad
\begin{array}{c}
\forall b, E; \rho \vdash_{ps} \mathbf{Vis} \ e_1 \ k_1 \approx_{\mathcal{R}}^{\ell} k_2(b) \\
e_1 : E \ A \quad e_2 : E \ B \\
\mathit{empty}(A) \quad \rho(e_1) \not\sqsubseteq \ell \quad \rho(e_2) \not\sqsubseteq \ell \\
\hline
[\text{EMPVISVISL}] \frac{}{E; \rho \vdash_{ps} \mathbf{Vis} \ e_1 \ k_1 \approx_{\mathcal{R}}^{\ell} \mathbf{Vis} \ e_2 \ k_2}
\end{array}$$

■ **Figure 6** Inference rules for indistinguishability, where events are halting and not visible.

PRIVVISINDL is analogous to TAUL, but for secret events instead of τ nodes. This rule has the same premises as PRIVVISTAU for the same reasons. Moreover, it only removes a node from the head of one ITree, not both. As with the definition of **seutt**, TAUL, and TAUR, we therefore make PRIVVISINDL inductive, not coinductive, to avoid relating an infinite stream of secret events to all other ITrees.

Finally, PRIVVISVIS removes a private event from the head of both sides of the relation. As with the previous rules, we require both events to be private and have non-empty answer types. This time, we require the continuations of the two events to be indistinguishable for every possible response *of both events separately*. This requirement formalizes the idea that the adversary should not be able to distinguish the program's behavior on any pair of secret responses.

To see the power of this rule, consider whether an adversary who can see l but not h would find the following ITrees indistinguishable from themselves:

$$\begin{array}{ll}
t_{\text{sec}} \triangleq x \leftarrow \mathbf{trigger} \ \mathbf{get}(l); & t_{\text{insec}} \triangleq x \leftarrow \mathbf{trigger} \ \mathbf{get}(l); \\
y \leftarrow \mathbf{trigger} \ \mathbf{get}(h); & y \leftarrow \mathbf{trigger} \ \mathbf{get}(h); \\
\mathbf{trigger} \ \mathbf{set}(h, x + y) & \mathbf{trigger} \ \mathbf{set}(l, x + y)
\end{array}$$

One would hope that t_{sec} would be indistinguishable from itself, while t_{insec} would not be, and indeed that is the case. To (attempt to) prove that either tree is equivalent to itself, we walk through each ITree. Since l is visible, so is **get**(l), so PUBVIS applies and requires only that each possible value of x produce an ITree that is indistinguishable from itself. Because h is secret, the adversary should not be able to observe or infer its value, so we must use PRIVVISVIS to remove **get**(h). PRIVVISVIS requires that, for all possible *pairs* of values y_1, y_2 , the continuations be indistinguishable. Thus in t_{sec} , **trigger set**($h, x + y_1$) must be indistinguishable from **trigger set**($h, x + y_2$). Since h is secret, so are the **set** events, so PRIVVISVIS can remove them even when they differ. After removing **set**, the remaining continuation always produces **ret** (), so RET finishes the proof.

However, in t_{insec} , PRIVVISVIS does not apply to the **set** events since l is visible. PUBVIS only relates ITrees starting with the same event, but **set**($l, x + y_1$) \neq **set**($l, x + y_2$) when $y_1 \neq y_2$. As a result, no rule applies after removing **get**(h), so the adversary can distinguish t_{insec} from itself. In other words, t_{insec} is, indeed, insecure.

Private Halting Events. Finally, we turn to the case where an event the adversary cannot see halts the computation. In this case, the adversary should be unable to tell that the event took place, and therefore should not be able to distinguish a program with a secret halt from a program that never terminates. However, the adversary should still be able to distinguish it from any ITree that contains an event the adversary can see.

This intuition means that a private halting event should not be treated like a τ , as a private non-halting event is, but rather should be indistinguishable from *an infinite stream of τ s*. We formalize this approach with the rules presented in Figure 6 along with their symmetric analogues. EMPVISTAU peels a single τ off the right ITree, leaving the private halting event on the left unmodified. EMPVISVISL does the same for a private event.

There are two interesting properties about these rules. First, unlike the rules for private events and τ s that leave one side of the equivalence unmodified, these rules are coinductive, not inductive. This choice allows us to relate a private halting event to an entire nonterminating program, as long as that program has no public events. Indeed, no rule allows us to remove a private halting event, as there would be nothing left to compare. Second, EMPVISVISL has no requirement that B , the answer type of the not-necessarily-halting event, be non-empty. This choice avoids the need to explicitly handle the case where both ITrees contain private halts. If B is non-empty, then EMPVISVISL treats the event as a τ . If B is empty, then the first premise of the rule is trivially satisfied, which is desirable, as in that case both ITrees begin with a private halt event and should be equivalent.

4.2 The Metatheory of Indistinguishability

The seutt relation captures intuitions about when two ITrees are indistinguishable to some adversary, but using it requires a delicate mix of induction and coinduction. To both demonstrate the power of our definition and better support verification, we also develop a library of metatheory for indistinguishability. This library supports reasoning about cross-language toolchains without the need for explicit coinduction, as we will see when we verify the correctness of a security type system and compiler for IMP (Sections 5 and 6, respectively).

Indistinguishability as a PER Model. Recall from Section 2.1 that Sabelfeld and Sands [46] argue for indistinguishability forming a partial equivalence relation (PER). It would be nice if seutt always formed a PER, but because it is parameterized on an arbitrary relation for return values, that is not always the case. Instead, we prove generalized versions of transitivity and reflexivity. In particular, if we let $\overleftarrow{\mathcal{R}}$ denote the reverse relation of \mathcal{R} – that is, $\overleftarrow{\mathcal{R}}(x, y) \triangleq \mathcal{R}(y, x)$ – then the following theorems hold.

► **Theorem 3.** *For all \mathcal{R} , E , ρ , and ℓ , if $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}}^{\ell} t_2$, then $E; \rho \vdash_{ps} t_2 \approx_{\overleftarrow{\mathcal{R}}}^{\ell} t_1$.*

► **Theorem 4.** *If $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}_1}^{\ell} t_2$ and $E; \rho \vdash_{ps} t_2 \approx_{\mathcal{R}_2}^{\ell} t_3$ then $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}_1 \circ \mathcal{R}_2}^{\ell} t_3$.*

Note that if \mathcal{R} is symmetric, then $\mathcal{R} = \overleftarrow{\mathcal{R}}$, and if \mathcal{R} is transitive, then $\mathcal{R} \circ \mathcal{R} \subseteq \mathcal{R}$. These properties allow us to prove the following corollary.

► **Corollary 5.** *If \mathcal{R} is a PER, then so is $E; \rho \vdash_{ps} - \approx_{\mathcal{R}}^{\ell} -$ for any E , ρ , and ℓ .*

ITree Combinators. ITrees are often defined using the combinators from Section 2.2, making it important to understand how indistinguishability interacts with those combinators. The definition of seutt directly describes how to relate simple programs defined using only ret and trigger , but they say nothing about larger ITrees built using bind and iteration .

bind allows for the sequential composition of programs. We would like indistinguishable programs t_1 and t_2 followed by indistinguishable continuations k_1 and k_2 to compose into larger indistinguishable programs $t_1 \gg k_1$ and $t_2 \gg k_2$. The following theorem says that this result holds whenever the relation \mathcal{R}_1 , securely relating t_1 and t_2 , puts enough constraints on their possible outputs to ensure that k_1 and k_2 are always securely related at some relation \mathcal{R}_2 .

► **Theorem 6.** *If $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}_1}^{\ell} t_2$ and for all values a, b , $\mathcal{R}_1(a, b)$ implies $E; \rho \vdash_{ps} k_1(a) \approx_{\mathcal{R}_2}^{\ell} k_2(b)$, then $E; \rho \vdash_{ps} t_1 \gg k_1 \approx_{\mathcal{R}_2}^{\ell} t_2 \gg k_2$.*

Iteration represents loops, which have two parts: an initial value, and a body that produces a value from the previous value. Indistinguishable initial values paired with indistinguishable bodies produce indistinguishable loops, as we can see in the following theorem.

► **Theorem 7.** *If $\mathcal{R}_1(a_1, b_1)$ and, for any $a, b, E; \rho \vdash_{ps} k_1(a) \approx_{\text{caseR}(\mathcal{R}_1, \mathcal{R}_2)}^\ell k_2(b)$ whenever $\mathcal{R}_1(a, b)$, then $E; \rho \vdash_{ps} \text{iter } k_1 \ a_1 \approx_{\mathcal{R}_2}^\ell \text{iter } k_2 \ b_1$.*

This rule is conceptually similar to a loop invariant from a Hoare-style logic. \mathcal{R}_1 is a property that is initially true and is preserved on each iteration except the final one, while the final iteration guarantees that \mathcal{R}_2 holds. The $\text{caseR}(\mathcal{R}_1, \mathcal{R}_2)$ function lifts two relations to a single relation over sum types such that \mathcal{R}_1 is applied to two left values, \mathcal{R}_2 is applied to two right values, and no other combination is related.

Relationship with Equivalence Up-To Taus. Recall that weak bisimulation of ITrees (eutt) requires two ITrees to contain the same pattern of interaction with their environment. Our notion of indistinguishability assumes that adversaries distinguish programs purely based on their interactions with the environment. One would thus expect that combining eutt with indistinguishability should result in indistinguishability. The following theorem shows this to be the case.

► **Theorem 8 (Mixed Transitivity).** *If both $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}_1}^\ell t_2$ and $E \vdash t_2 \approx_{\mathcal{R}_2} t_3$ then we can conclude that $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}_1 \circ \mathcal{R}_2}^\ell t_3$.*

This is a very powerful theorem. In particular, many program transformations preserve equality. That is, they take source programs with equivalent-up-to-taus ITree representations to target programs with the same property. Mixed transitivity tells us that compilers built from such transformations also preserve indistinguishability. For instance, since noninterference – the security property we are ultimately considering – is defined as a program being indistinguishable from itself, mixed transitivity supports a very simple proof that the compiler in Section 6 preserves noninterference. While this result might be surprising, it reflects the utility of ITrees and indistinguishability. By looking at which labels can distinguish an ITree from itself, we can discover where leaks are possible.

4.3 Progress-Insensitive Indistinguishability

The type systems that enforce progress-sensitive noninterference are extremely restrictive. Thus, information-flow control literature mostly studies progress-*insensitive* type systems. These type systems enforce noninterference against adversaries who cannot see when a program has begun to silently loop forever. Intuitively, such adversaries believe that silently looping programs could break out of their loops at any moment, and so do not distinguish them from programs which have produced visible events.

In order to support such reasoning, we introduce pi-seutt , a progress-insensitive version of indistinguishability for ITrees. This leads to the following definition:

► **Definition 9 (pi-seutt).** *The relation pi-seutt , the progress-insensitive version of indistinguishability, is defined by modifying the definition of seutt by completely removing the rules for halting events (all rules in Figure 6) and making every other rule coinductive (this modifies TAUL and TAUR in Figure 4 as well as PRIVVISINDL in Figure 5 and its not-presented symmetric counterpart).*

This relation is strictly more permissive than seutt , since it relates every ITree to silently diverging ITrees and private halts. These facts can be formalized in the following theorems:

► **Theorem 10.** *If $E; \rho \vdash_{ps} t_1 \approx_{\mathcal{R}}^{\ell} t_2$ then $E; \rho \vdash_{pi} t_1 \approx_{\mathcal{R}}^{\ell} t_2$.*

► **Theorem 11.** *Given any ITree t , $E; \rho \vdash_{pi} t_{spin} \approx_{\mathcal{R}}^{\ell} t$.*

► **Theorem 12.** *Given any ITree t , if e is a halting event, then $E; \rho \vdash_{pi} \text{Vis } e \ k \approx_{\mathcal{R}}^{\ell} t$.*

Just as with the progress-sensitive version of indistinguishability, we can show that indistinguishability plays well with the usual ITree combinators. This allows us to prove ITrees indistinguishable in many cases without resorting to hand-rolled coinduction.

► **Theorem 13.** *If $E; \rho \vdash_{pi} t_1 \approx_{\mathcal{R}_1}^{\ell} t_2$ and $E; \rho \vdash_{pi} k_1(a) \approx_{\mathcal{R}_2}^{\ell} k_2(b)$ whenever $\mathcal{R}_1(a, b)$, then $E; \rho \vdash_{pi} t_1 \gg_{\mathcal{R}_1}^{\ell} k_1 \approx_{\mathcal{R}_2}^{\ell} t_2 \gg_{\mathcal{R}_2}^{\ell} k_2$.*

► **Theorem 14.** *If $\mathcal{R}_1(a_1, a_2)$ and for any a, a' , $E; \rho \vdash_{pi} k_1(a) \approx_{\text{caseR}(\mathcal{R}_1, \mathcal{R}_2)}^{\ell} k_2(a')$ whenever $\mathcal{R}_1(a, a')$, then $E; \rho \vdash_{pi} \text{iter } k_1 \ a_1 \approx_{\mathcal{R}_2}^{\ell} \text{iter } k_2 \ a_2$.*

Moreover, mixed transitivity again holds, allowing for simple proofs of compiler safety:

► **Theorem 15 (Mixed Transitivity).** *If both $E; \rho \vdash_{pi} t_1 \approx_{\mathcal{R}_1}^{\ell} t_2$ and $E \vdash t_2 \approx_{\mathcal{R}_2} t_3$ then we get $E; \rho \vdash_{pi} t_1 \approx_{\mathcal{R}_1 \circ \mathcal{R}_2}^{\ell} t_3$.*

Progress-insensitive indistinguishability behaves differently from the progress-sensitive sibling version in one important way: it does not form a PER. Because it relates a diverging ITree to every other ITree, `pi-seutt` is not transitive. This is not surprising, since progress-insensitive indistinguishability is not a PER [54, 42, 16]. It does, however, retain generalized symmetry, and a weakened but still-useful version of generalized transitivity:

► **Theorem 16.** *If $E; \rho \vdash_{pi} t_1 \approx_{\mathcal{R}}^{\ell} t_2$ then $E; \rho \vdash_{pi} t_2 \approx_{\mathcal{R}}^{\ell} t_1$.*

► **Theorem 17.** *If $E; \rho \vdash_{pi} t_1 \approx_{\mathcal{R}_1}^{\ell} t_2$, $E; \rho \vdash_{pi} t_2 \approx_{\mathcal{R}_2}^{\ell} t_3$, and t_2 converges along all paths, then $E; \rho \vdash_{pi} t_1 \approx_{\mathcal{R}_1 \circ \mathcal{R}_2}^{\ell} t_3$.*

Where an ITree is considered convergent if it is either a `ret`, a τ followed by a convergent ITree, or a non-halting event followed by a continuation that converges for any input.

Unlike progress-sensitive indistinguishability, we can easily show that loops produce no events that are observable to some adversary at ℓ via `pi-seutt`. Suppose that we want to show that `iter` body a_0 emits no events that are observable to some adversary at ℓ . We can do so by showing that `iter` body a_0 and `ret` b are indistinguishable with some return relation \mathcal{R} . This shows that the body of the loop both emits no observable events and, if the loop terminates, it returns a value c where $\mathcal{R}(c, b)$. Importantly, we have not made any statement about whether the loop terminates; we have merely said that it will not produce events, regardless of its termination behavior. We formalize this in the following theorem:

► **Theorem 18.** *For any relation \mathcal{R}_{inv} , if*

$$\mathcal{R}_{inv}(a_0, b) \quad \text{and} \quad \forall a, \mathcal{R}_{inv}(a, b) \implies E; \rho \vdash_{pi} \text{body } a \approx_{\text{leftcase}(\mathcal{R}_{inv}, \mathcal{R})}^{\ell} \text{ret } b,$$

then $E; \rho \vdash_{pi} \text{iter } \text{body } a_0 \approx_{\mathcal{R}}^{\ell} \text{ret } b$, where the relation `leftcase` is defined as follows:

$$\text{leftcase}(\mathcal{R}_1, \mathcal{R}_2)(\text{inl}(a), b) = \mathcal{R}_1(a, b) \quad \text{leftcase}(\mathcal{R}_1, \mathcal{R}_2)(\text{inr}(a), b) = \mathcal{R}_2(a, b)$$

4.4 Noninterference and Interpretation

Recall from Section 2.1 that we can define noninterference using an indistinguishability relation on programs by saying that a program is noninterfering if it is related to itself – given indistinguishable inputs, it will produce indistinguishable computations. We could define noninterference on ITrees using `seutt` (or `pi-seutt`), as they provide such indistinguishability relations by design. This approach produces a sensible definition, but one that assumes an extremely strong adversary.

Consider the following IMP program, where the h_i s have label ℓ_h and the l_i s have label ℓ_l :

```
if ( $h_1 = 0$ ) then  $\{h_2 := l_1\}$  else  $\{h_2 := l_2\}$ 
```

Since the program writes only to secret variables, intuitively this program seems secure. However, according to `seutt`, it is not related to itself at ℓ_l since reading from l_1 and l_2 produce different `get` events with label ℓ_l . All adversaries have the power to observe *reads* of public state, not just writes.

The visibility of public read events is not the only problem. Using just `seutt` also means a computation cannot publicly depend on the result of reading a secret variable, even if a public value were written to that variable. For instance, the following program would also be considered insecure:

```
 $h := l$ ; print( $\ell_l, h$ )
```

If h cannot change between assignments, this program is intuitively secure, but `seutt` at ℓ_l requires `print`(ℓ_l, h) to produce the same output regardless of the value of h , which it clearly does not.

On uninterpreted ITrees, `seutt` models a system where both reads and writes are visible to anyone who can see the variable, and the value of a secret variable may silently change between a read and a write. This model makes perfect sense in some contexts – like distributed computation [27] – but we usually consider weaker adversaries.

We can remove these assumptions and model a weaker adversary by interpreting state, as we discussed in Section 2.4. Interpreting these programs would result in two meta-level functions (i.e., Coq functions) which take a state as input and produce an ITree returning an output state. For example in Section 2.4, we define the semantics of an IMP program c as an interpreted ITree – that is, as a function from states to ITrees – not as a single ITree with state events. We thus adjust our notions of indistinguishability and noninterference to account for this semantic construct.

Intuitively, we start with a family of relations $\mathcal{R}_{S,\ell}$ that describes when states are indistinguishable to an adversary at level ℓ and use it to define the following observational equivalence. For technical reasons, we require $\mathcal{R}_{S,\ell}$ to be an equivalence relation at all labels. For IMP, we use a relation \cong_1^ℓ which only requires states to agree on a variable x if the label of x flows to ℓ .

► **Definition 19** (Stateful Indistinguishability). *Two stateful computations p_1 and p_2 are px-statefully indistinguishable under $\mathcal{R}_{S,\ell}$ and \mathcal{R} at label ℓ if, for every pair of states σ_1 and σ_2 such that $\mathcal{R}_{S,\ell}(\sigma_1, \sigma_2)$,*

$$E; \rho \vdash_{px} p_1 \sigma_1 \approx_{\mathcal{R}_{S,\ell} \times \mathcal{R}}^{\ell} p_2 \sigma_2$$

$$\text{where } \mathcal{R}_{S,\ell} \times \mathcal{R}((\sigma'_1, a_1), (\sigma'_2, a_2)) \stackrel{\Delta}{\iff} \mathcal{R}_{S,\ell}(\sigma'_1, \sigma'_2) \text{ and } \mathcal{R}(a_1, a_2)$$

$$\frac{\Gamma(x) \sqsubseteq \ell}{\Gamma \vdash x : \ell} \qquad \frac{}{\Gamma \vdash n : \ell} \qquad \frac{\Gamma \vdash e_1 : \ell_1 \quad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \odot e_2 : \ell_1 \sqcup \ell_2}$$

■ **Figure 7** Typing rules for expressions in security-typed IMP.

As described above, stateful indistinguishability with \cong_{Γ}^{ℓ} defines security against an adversary who can observe public writes, but not secret writes or secret reads. This indistinguishability relation leads to a much more common definition of noninterference, and it is the one we will use in our case studies in Sections 5 and 6.

► **Definition 20** (Noninterference). *A stateful computation is px-noninterfering with state relations $\mathcal{R}_{S,\ell}$ and return relation \mathcal{R} if, given any label ℓ , it is px-statefully indistinguishable from itself under state relation family $\mathcal{R}_{S,\ell}$ and return relation \mathcal{R} .*

5 Security Sensitive Type Systems For Imp

To see how to use this theory of indistinguishability and ITrees, we now provide an information-security guarantee for an example toolchain for IMP. We begin by verifying two information-flow type systems, and proceed with a simple compiler in Section 6. The two notions of noninterference – progress sensitive and progress insensitive – require slightly different type systems, so we use our ITrees-based semantics to formally verify that both enforce their respective notions of noninterference. As is common in such type systems, we assume \mathcal{L} forms a join semilattice with a unique least element \perp representing “completely public.”

5.1 Two Type Systems

Both type systems have two typing judgments: one for expressions and one for commands. The typing judgments for expressions take the form $\Gamma \vdash e : \ell$, where Γ is a map from variables to information flow labels, and ℓ is a label. The judgment says that e is well-typed and depends only on information at or below label ℓ . The typing rules for expressions, which are the same for both type systems, are presented in Figure 7.

The typing rules for commands are presented in Figure 8. As these rules differ between the progress-sensitive and progress-insensitive type systems, we annotate the turnstyles with ps for progress-sensitive rules, pi for progress-insensitive rules, and px for rules that are identical in both type systems.

The typing judgments for commands take the form $\Gamma; pc \vdash_{px} c \diamond \ell_{ex}$, where pc and ℓ_{ex} are information-flow labels. The pc label is a *program-counter label* that tracks the sensitivity of the control flow, while the second label ℓ_{ex} is an upper bound on the label of any exceptions c might raise. Note that the rules listed in Figure 8 do not include any way to type check an inlined ASM program. We address this concern in Section 5.3.

Program-counter labels are a standard technique to control *implicit information flows* – that is, information leaked by the control flow [45]. For example, consider the following program where h has label ℓ_h and l has label ℓ_l with $\ell_h \not\sqsubseteq \ell_l$:

if ($h = 0$) then $\{l := 0\}$ else $\{l := 1\}$

While l is only ever explicitly set to constant values, its final value clearly depends on the secret h . The pc label allows us to detect and eliminate these flows by tracking the sensitivity of the control flow. Specifically, the IF rule requires the condition’s label to flow to the pc in

Shared Typing Rules

$[\text{SKIP}] \frac{}{\Gamma; pc \vdash_{px} \text{skip} \diamond \perp}$ $[\text{ASSIGN}] \frac{\Gamma \vdash_{px} e : \ell \quad pc \sqcup \ell \sqsubseteq \Gamma(x)}{\Gamma; pc \vdash_{px} x := e \diamond \perp}$ $[\text{TRY}] \frac{\Gamma; pc \vdash_{px} c_1 \diamond \ell_{ex} \quad \Gamma; pc \sqcup \ell_{ex} \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} \text{try } \{c_1\} \text{ catch } \{c_2\} \diamond \ell'_{ex}}$	$[\text{IF}] \frac{\Gamma \vdash_{px} e : \ell \quad \Gamma; pc \sqcup \ell \vdash_{px} c_1 \diamond \ell_{ex} \quad \Gamma; pc \sqcup \ell \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} \text{if } (e) \text{ then } \{c_1\} \text{ else } \{c_2\} \diamond \ell_{ex} \sqcup \ell'_{ex}}$ $[\text{SEQ}] \frac{\Gamma; pc \vdash_{px} c_1 \diamond \ell_{ex} \quad \Gamma; pc \sqcup \ell_{ex} \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} c_1 ; c_2 \diamond \ell_{ex} \sqcup \ell'_{ex}}$ $[\text{PRINT}] \frac{\Gamma \vdash_{px} e : \ell \quad pc \sqcup \ell \sqsubseteq \ell'}{\Gamma; pc \vdash_{px} \text{print}(e, \ell') \diamond \perp}$
<p style="text-align: center;">Progress-Sensitive Typing Rules</p> $[\text{WHILE-PS}] \frac{\Gamma \vdash_{ps} e : \perp \quad \Gamma; \perp \vdash_{ps} c \diamond \perp}{\Gamma; \perp \vdash_{ps} \text{while } (e) \text{ do } \{c\} \diamond \perp}$ $[\text{THROW-PS}] \frac{}{\Gamma; \perp \vdash_{ps} \text{throw}(\perp) \diamond \perp}$	<p style="text-align: center;">Progress-Insensitive Typing Rules</p> $[\text{WHILE-PI}] \frac{\Gamma \vdash_{pi} e : \ell \quad \Gamma; pc \sqcup \ell \sqcup \ell_{ex} \vdash_{pi} c \diamond \ell_{ex}}{\Gamma; pc \vdash_{pi} \text{while } (e) \text{ do } \{c\} \diamond \ell_{ex}}$ $[\text{THROW-PI}] \frac{pc \sqsubseteq \ell_{ex}}{\Gamma; pc \vdash_{pi} \text{throw}(\ell_{ex}) \diamond \ell_{ex}}$

■ **Figure 8** Typing rules for commands in security-typed IMP.

each branch, and the ASSIGN rule requires the pc to flow to the label of the variable being assigned. In the above example, the label of the condition $h = 0$ is ℓ_h , so IF requires c_1 and c_2 to type check with a pc where $\ell_h \sqsubseteq pc$. Since $\Gamma(l) = \ell_l$, ASSIGN requires $pc \sqsubseteq \ell_l$. Transitivity of \sqsubseteq thus requires $\ell_h \sqsubseteq \ell_l$, which it does not, so the program correctly fails to type check.

Exceptions can affect the control flow of a program, and therefore can also cause implicit flows of information. Consider the following program.

if ($h = 0$) then {throw(ℓ_h)} else {skip}; $l := 1$

Much like the previous example, this program only assigns l to a constant, yet it still leaks the value of h . We use a standard technique [32, 40] that relies on exception labels in the typing judgment. As previously mentioned, the exception label of a program c is an upper bound on the labels of any exception c might raise. To eliminate exception-based leaks, the SEQ rule increases the pc label of the second command by the exception label of the first. The TRY rule makes similar use of the exception label, increasing the pc in the catch block, as that command only executes if an exception is thrown.

The SKIP rule is simple, as skip can never have an effect. PRINT produces a flow of information to an output channel labeled ℓ' , so it checks that ℓ' may safely see both the expression being written and the fact that this command executed.

The rules for while loops and throw statements are different for the progress-sensitive and progress-insensitive type systems, so we handle them separately.

Progress-Sensitive While and Throw Rules. In a progress-sensitive setting, the adversary can observe nontermination. As a result, a program's termination behavior can only safely depend on completely public information. WHILE-PS enforces this requirement in a standard,

but highly restrictive way [55]: the loop condition and the pc of the context must both be the fully public label \perp . Moreover, any exceptions thrown in the body of the loop could also influence termination behavior, so those must be fully public as well.

Recall from Section 4 that a low observer cannot distinguish between an uncaught secret exception and an infinite loop. Thus non-public exceptions create the same implicit flows as while loops, so THROW-PS restricts exceptions in much the same way as WHILE-PS restricts loops: everything must be fully public.

Progress-Insensitive While and Throw Rules. In a progress-insensitive setting, the adversary cannot see nontermination, so secrets can safely influence the termination behavior of a program. The WHILE-PI rule therefore allows loops with any pc . Since both the loop condition and any exceptions the loop body throws influence whether the body is run, WHILE-PI increases the pc in the loop body by both the loop guard label and the body's exception label.

For the same reason, THROW-PI is more permissive than its progress-sensitive counterpart. In particular, the label on the exception just needs to be at least as secret as the pc label.

5.2 Proving Security

Both type systems enforce their respective notions of noninterference (Definition 20). Unlike many existing proofs of noninterference, our proofs using ITrees proceed by simple induction over the syntax of IMP. This simplicity is made possible by the combination of two facts: our IMP semantics is given by simple induction using ITrees combinators, and those combinators interact with indistinguishability in predictable ways, as described by the metatheory of Section 4.

Type systems are inherently compositional: we are able to conclude that a program is secure knowing nothing about subprograms other than that they also type check. However, our semantic definition of noninterference is *not* fully compositional. To see this, consider the IMP program $p = l := h ; \text{throw}(\ell)$. This program updates the state in an insecure way, assigning a high-security value to a low-security variable, and then throws a low-security exception. In fully interpreted programs, the updated state is part of the return value, but adversaries cannot observe that return value if an exception is thrown (see Section 3), making p semantically secure. However, if we catch the exception, the adversary once again can see the effect of the assignment $l := h$. Thus, p does not compose securely.

In order for our type system to enforce security compositionally, it enforces two properties beyond noninterference. Each rules out programs which, like p above, are secure but do not compose securely. The first describes how state and exceptions interact in a secure setting, which will rule out the example program above. The second, called *confinement*, defines how effects are bound by the type system.

Interaction of Exceptions and State. Our first goal is to semantically rule out programs like p above, allowing us to reason compositionally about exception handlers. In order to do so, we need to reason about what state updates are performed before an exception is thrown. However, since in our semantics of IMP we interpret state events while leaving exceptions as ITree events, the result state of an IMP program is forgotten when an exception is thrown.

This correctly models our adversary, who cannot distinguish between private exceptions and silently diverging programs. But in order to achieve compositionality, we need to keep information about the final state before an exception is raised. We accomplish this with a condition on an alternative semantics for IMP programs. In this semantics, exceptions are

interpreted into the standard sum type representation before state events are interpreted. This interpretation, $\text{interp } h_{\text{prog}} (\text{interp } h_{\text{exc}} \llbracket c \rrbracket_c)$, is a stateful function that returns a final state along with either a result of type `unit` or the label of an exception. We can inspect this final state to ensure that the program always takes indistinguishable states to indistinguishable states.

We formalize this property as follows, where the relation \cong_{\top}^{ℓ} requires that states agree on a variable x only when $\Gamma(x) \sqsubseteq \ell$, as in Section 4.4.

► **Definition 21** (Exceptions-and-State Property). *A command c satisfies the px-exceptions-and-state property if $\text{interp } h_{\text{prog}} (\text{interp } h_{\text{exc}} \llbracket c \rrbracket_c)$ is statefully indistinguishable from itself under \cong_{\top}^{ℓ} and \top at every label ℓ .*

Note the use of \top as the output relation means we ignore whether or not c threw an exception, while we still ensure that the final states are indistinguishable. Ignoring this information in this property is acceptable because it is captured by our standard noninterference condition.

Confinement. Even with the exceptions-and-state property, implicit flows, like the motivating our use of pc labels, can still break compositionality. Confinement fixes this.

In the typing judgment for commands, the pc and ℓ_{ex} labels are both designed to constrain effects. If a command type checks with pc and ℓ_{ex} , it should have no effects visible *below* pc and no (uncaught) exceptions *above* ℓ_{ex} . Semantically, a program has no visible effects below pc if, for any label ℓ where $pc \not\sqsubseteq \ell$, it is indistinguishable from `skip`. For any uncaught exception terminating a ITree, we simply check that the exception's label flows to ℓ_{ex} . We formalize this idea into the following property called *confinement*.

► **Definition 22** (Confinement). *A command c is px-confined to pc with ℓ_{ex} exceptions, if, for all labels ℓ such that $pc \not\sqsubseteq \ell$, the following conditions hold.*

1. c is indistinguishable from `skip` at ℓ : $\text{interp } h_{\text{prog}} \llbracket c \rrbracket_c$ and $\text{interp } h_{\text{prog}} \llbracket \text{skip} \rrbracket_c$ are px -statefully indistinguishable under \cong_{\top}^{ℓ} and $=$ at ℓ .
2. c makes no modifications to the state visible at ℓ : $\text{interp } h_{\text{prog}} (\text{interp } h_{\text{exc}} \llbracket c \rrbracket_c)$ and $\text{interp } h_{\text{prog}} (\text{interp } h_{\text{exc}} \llbracket \text{skip} \rrbracket_c)$ are px -statefully indistinguishable under \top and $=$ at ℓ .
3. For all initial state heap states h and register states r where c throws an exception, the label of that exception flows to ℓ_{ex} :

$$E \vdash (\text{interp } h_{\text{prog}} (\text{interp } h_{\text{exc}} \llbracket c \rrbracket_c))(r, h) \approx_{=} \text{ret } (r', h', \text{inr}(\ell'_{\text{ex}})) \implies \ell'_{\text{ex}} \sqsubseteq \ell_{\text{ex}}$$

Together, these definitions restrict programs to those that compose securely, as required by the type system. With this compositionality property, we can prove that our type system enforces the conjunction of all three properties.

► **Theorem 23.** *If $\Gamma; pc \vdash_{px} c \diamond \ell_{\text{ex}}$, then c is px -noninterfering (Definition 20), satisfies the px -exceptions-and-state property, and is px -confined to pc with ℓ_{ex} exceptions.*

5.3 Semantic Typing and Inline Asm

Both type systems above enforce security, but are highly conservative. Many secure programs fail to type check, notably including any secure program with inlined ASM. To support our goal of cross-language security reasoning and address this concern without the need to introduce a type system for ASM, we provide a *semantic typing* [21] rule.

One would hope that the three conditions discussed above would be sufficient. However, the possibility of undefined ASM behavior (see Section 2.5) necessitates an additional condition. We thus introduce the notion of *inline validity*, which requires inlined ASM to depend only on the initial heap state, not the initial register state, thereby ruling out undefined behavior.

► **Definition 24** (Inline Validity). *An ASM program a is inline-valid if, given any two register states r_1 and r_2 , and any heap states h , then a run with (r_1, h) and (r_2, h) produces the same changes to the heap. That is, if $p = \text{interp } h_{\text{prog}} (\text{interp } h_{\text{exc}} \llbracket a \rrbracket_{\text{asm}})$, then*

$$\text{printE} \vdash p(r_1, h) \approx_{\top \times =} p(r_2, h).$$

Note that any ASM program that only ever reads from a register after it has written to that register will satisfy this property. We also lift this definition to whole IMP programs by applying it separately to each inlined ASM block.

► **Definition 25** (Validity). *c is a valid IMP program if any inlined ASM program it contains is an inline-valid ASM program.*

Including validity with our other semantic conditions is sufficient to guarantee security, so we can safely define the following semantic typing rule.

$$\begin{array}{c} c \text{ is px-noninterfering} \\ c \text{ satisfies the px-exceptions-and-state property} \\ c \text{ is px-confined to } pc \text{ and } \ell_{ex} \\ c \text{ is valid (Definition 25)} \\ \hline \text{[SEMANTIC]} \frac{}{\Gamma; pc \vdash_{px} c \diamond \ell_{ex}} \end{array}$$

Adding this new rule to both type systems allows them to reason about multi-language programs including inline ASM and larger systems, even when the syntactic type system cannot reason about every component. Importantly, SEMANTIC (??) is sound from a security perspective. That is, Theorem 23 continues to hold for both extended type systems.

6 Preserving Noninterference Across Compilation

For a compiled language like IMP, noninterference is only part of the story. After all, rather than run IMP code directly, programmers instead compile IMP to ASM and run the ASM. Compilation can change programs significantly, and can introduce insecurity in the process. Thus, we need to ensure that the compiler translates noninterfering IMP programs into noninterfering ASM programs. We now turn our attention to the proof-engineering effort involved in providing such an assurance. In particular, we show that (a) adding exceptions and information-flow labels to IMP does not complicate the proof of compiler correctness, and (b) turning a proof of correctness into a proof of noninterference preservation is simple using mixed transitivity (Theorem 8).

Note that, to build our compiler, we had to fix the number of information-flow labels. We thus specialize our discussion of IMP from Section 5 to the two-point lattice $\mathcal{L} = \{\top, \perp\}$. Using any other finite lattice would require only minimal changes.

6.1 Asm, Its Semantics, and the Compiler

Figure 9 presents the syntax of ASM, the simple assembly language that our compiler targets. An ASM program is a sequence of *blocks*, where each block starts at some address A and consists of a sequence of straight-line instructions followed by a single jump. The first block must be at the special address `START`.

Registers	r	$::=$	$\$0 \mid \$1 \mid \dots$
Operands	o	$::=$	$r \mid n$
Instructions	i	$::=$	$\text{ADD } r_1 \leftarrow r_2, o \mid \text{SUB } r_1 \leftarrow r_2, o \mid \text{MUL } r_1 \leftarrow r_2, o$ $\mid \text{EQ } r_1 \leftarrow r_2, o \mid \text{LEQ } r_1 \leftarrow r_2, o \mid \text{NOT } r \leftarrow o$ $\mid \text{MOV } r_1 \leftarrow r_2 \mid \text{LOAD } r \leftarrow x \mid \text{STORE } x \leftarrow r \mid \text{print}(\ell, r)$
Branches	b	$::=$	$\text{JMP } A \mid \text{BRZ } r \ A1 \ A2 \mid \text{RAISE } \ell$
Blocks	B	$::=$	$A : i_1 ; \dots ; i_n ; b$
Programs	p	$::=$	$\text{START} : i_1 ; \dots ; i_n ; b$ $B_1 ; \dots ; B_m$

■ **Figure 9** Secure ASM syntax where x is a variable, A is an address, n is a natural number, and ℓ is an information-flow label.

Most ASM instructions write to exactly one register, computing the written value from a combination of other registers and integer constants. For instance, $\text{ADD } \$0 \leftarrow \$1, 1$ takes the value of register $\$1$, adds one, and stores the result in register $\$0$. The MOV instruction copies the value of one register into another, while LOAD and STORE move information between registers and the heap. Finally, the PRINT instruction prints information to a stream, depending on the label ℓ .

Jumps are either direct jumps, conditional jumps, or exceptions. A direct jump $\text{JMP } A$ immediately moves execution to the beginning of the block with address A . A conditional jump $\text{BRZ } r \ A1 \ A2$ move execution to $A1$ if register r contains zero and $A2$ otherwise. The $\text{RAISE } \ell$ branch raises an exception. Note that there is no equivalent of catching an exception. We assume that ASM programs always jump to either the address of one of the program's blocks or a special EXIT address.

Rather than representing ASM syntax directly in our Coq code, we take a more compositional approach and represent *sub-Control-Flow Graphs (sub-CFGs)*. These represent the structure of part of an ASM program. While a complete ASM program contains a unique START address, sub-CFGs may contain multiple addresses accessible to the outside. We refer to addresses which are accessible to the outside as *input* addresses. Likewise, sub-CFGs may jump to undefined addresses, whereas complete ASM programs always jump either to a defined address or EXIT . We refer to the undefined addresses a sub-CFG may jump to as its *output* addresses. Thus, a complete ASM program is a sub-CFG with exactly one input address (START) and exactly one output address (EXIT).

Intuitively, sub-CFGs execute starting at some input address, potentially jumping internally several times before they jump to some output address. To represent this pattern, we give sub-CFGs semantics as functions from an address to an ITree that return an address. That is, the semantics of a sub-CFG takes as input the input address at which to start executing, and produces an ITree that returns the output address the program jumps to. This structure is due to Xia et al. [57], and their semantic needed only minor changes to accommodate printing and exception-throwing.

In Xia et al.'s original compiler, IMP code always mapped to complete ASM programs. However, to accommodate exception throwing, our compiler has an extra step of indirection. We map IMP programs to sub-CFGs with exactly one input address but *three* output addresses. The first represents EXIT , as in a complete ASM program, while the second two represent the location of exception handler code. Thus, we compile $\text{throw}(\ell)$ to a jump to the second address if $\ell = \perp$ and the third address if $\ell = \top$. To compile a try-catch command, we place one copy of the handler at the second address and a second copy at the third address. That

means any exception will jump to the handler code, regardless of the label of the exception, matching the semantics we gave IMP in Section 3. Note that we still need separate addresses for each label to properly compile *uncaught* exceptions.

For inlined ASM code, we would hope to include it in the compiled code directly with no changes. Unfortunately, if inlined ASM throws an exception with a RAISE instruction, the surrounding IMP code can catch it, but embedding the RAISE unmodified in the compiled output would render the exception uncatchable. To support catching these exceptions, we process inlined ASM to replace RAISE instructions with jumps to the appropriate address. This change causes the inlined exception to properly jump to the handler code.

While the infrastructure described above translates IMP code into sub-CFGs, the end goal of our compiler is to translate complete IMP programs into complete ASM programs. The final step uses the two output addresses for exceptions by linking the sub-CFG of the complete IMP program with *two different* handlers. The low-security exception handler raises a low-security exception, while the high-security exception handler raises a high-security exception. Thus, any IMP code that raises an exception compiles to a complete ASM program that raises that same exception, while IMP code that catches an exception compiles to a complete ASM program with equivalent control flow.

6.2 Compiler Correctness

We adapt Xia et al.’s [57] proof of compiler correctness to account for the modifications we have made to IMP and ASM. We formalize correctness by comparing the source and the target programs – after interpretation – using weak bisimilarity. Intuitively, two stateful programs are weakly bisimilar if, whenever they are given *related* start states, the resulting ITrees are weakly bisimilar. We use a return relation \mathcal{R}_{env} . \mathcal{R}_{env} ignores the register files and compares heaps using a relation \cong , which ensures that they map equal variables to equal values. We can now state the correctness theorem for the `compile` function.

► **Theorem 26.** *For any initial heap states h_1, h_2 such that $h_1 \cong h_2$, any register states r_1, r_2 , and a valid IMP command c , the following equation holds*

$$\text{excE} \oplus \text{printE} \vdash \text{interp } h_{imp} \llbracket c \rrbracket_c (r_1, h_1) \approx_{\mathcal{R}_{env}} \text{interp } h_{asm} \llbracket \text{compile}(c) \rrbracket_{asm} (r_2, h_2)$$

where $\mathcal{R}_{env}((_, h_1, _), (_, h_2, _)) \iff h_1 \cong h_2$.

Notably, the changes necessary to adapt Xia et al.’s [57] proof of correctness to our modified compiler are small and isolated. Most cases of the inductive proof, corresponding to existing language features, needed only cosmetic changes. The new language features required new, but conceptually uninteresting, cases.

6.3 Compiler Security

We finally turn to our ultimate goal: proving that our compiler preserves security. There are two important notions of security for our compiler, both of which require cross-language reasoning. The first is that secure source programs are indistinguishable – by all adversaries – from target programs. This property directly relates an IMP program to an ASM program. The second is that the compiler preserves noninterference. While noninterference itself is a property of a single program, *preserving* noninterference is a property of a translation between two languages, which requires cross-language reasoning.

In order to formalize the idea of a secure IMP program being indistinguishable from its compilation, we need to compare these programs, even though they come from different languages. Because we defined `seutt` purely semantically, we can use it as easily as if we

were comparing programs in the same language. We use the return relation \mathcal{R}_Γ^ℓ , which again ignores the register file and ensures that they map equal *visible* variables to equal values. The theorem then takes the following form.

► **Theorem 27.** *For any valid IMP program c , if $\text{interp } h_{\text{prog}} \llbracket c \rrbracket_c$ is noninterfering with state relation \mathcal{R}_Γ^ℓ and return relation $=$, and c is a valid IMP program, then the following **seutt** equation holds for any label ℓ , arbitrary register states r_1, r_2 and heap states h_1, h_2 such that $h_1 \cong_\Gamma^\ell h_2$.*

$$\text{excE} \oplus \text{printE} \vdash_{\text{px}} \text{interp } h_{\text{prog}} \llbracket c \rrbracket_c (r_1, h_1) \approx_{\mathcal{R}_\Gamma^\ell} \text{interp } h_{\text{prog}} \llbracket \text{compile}(c) \rrbracket_{\text{asm}} (r_2, h_2)$$

Our second theorem is simply that our compiler takes noninterfering IMP programs to noninterfering ASM programs.

► **Theorem 28 (Noninterference Preservation).** *For a valid IMP program c , if $\text{interp } h_{\text{prog}} \llbracket c \rrbracket_c$ is noninterfering with state relations \mathcal{R}_Γ^ℓ and return relation $=$, then the same holds for its compilation. That is, $\text{interp } h_{\text{prog}} \llbracket \text{compile}(c) \rrbracket_{\text{asm}}$ is noninterfering with \mathcal{R}_Γ^ℓ and $=$. This result holds for both progress-sensitive and progress-insensitive noninterference.*

Notably, the proofs of both theorems follows directly from Theorem 26 and mixed transitivity, showing the utility of mixed transitivity for cross-language security reasoning.

7 Related Work

Goguen and Meseguer [15] introduced noninterference to formalize confidentiality; that is, the intuitive notion that secret data does not leak to an adversary. Volpano et al. [56] enforce progress-insensitive noninterference with a type system, and Volpano and Smith [55] modify the type system to be progress-sensitive. These results led to a long line of work introducing noninterference to an increasing complicated settings [40, 32, 45, 64, 33, 61, 41, 30, 53, 4, 44, 51, 1]. Proving the security of these varied type systems led to complicated arguments for noninterference, but also gave rise to an informal library of proof techniques. This work fits into a tradition of proof techniques for noninterference via models.

Most models view noninterference either as a trace (hyper)property or as the result of an indistinguishability relation. These perspectives are not mutually exclusive; we can view two programs as indistinguishable if they produce equivalent traces. Their focus, however, can be quite different. Trace-based models view noninterference as a 2-safety hyperproperty [12]. That is, noninterference can be falsified using finite prefixes of two traces. Specifically, for any interfering program there are two inputs that differ only on secrets but produce distinguishable events after a finite number of steps.

Indistinguishability models focus more on building compositional relations. Pioneered by Abadi et al. [1] and Sabelfeld and Sands [46], these models use PERs and define secure programs as those that are self-related. Two such approaches have yielded recent notable results. First, logical-relations techniques [43] inductively assign each type a binary relation. By constructing the relation to reflect the security requirements of the type, logical relations can reason about information flow control and noninterference [54, 42, 16]. Second, bisimulation approaches directly match up program executions to define indistinguishability [48, 13].

This work straddles these methods. ITrees intuitively collect all possible traces of a program into one infinite data structure. Our binary indistinguishability relation on ITrees is thus combining the hyperproperty model of noninterference with the indistinguishability model. Moreover, our indistinguishability relation is built on top of weak bisimulation. To give meaning to a type system, we also build a small logical relation connecting types to our bisimulation arguments.

To remain practical, many languages provide only progress-insensitive guarantees [28, 27, 56, 40], despite the fact that termination channels alone can leak arbitrary amounts of data [6]. Techniques for enforcing progress-sensitive guarantees [55, 45] exist, but have seen little use. Recent work attempts to unify the two by explicitly considering termination leaks as declassifications [11]. Like other models of noninterference [16], `seutt` is naturally progress-sensitive, giving a strong guarantee. We include the progress-insensitive `pi-seutt` to give ITree-based semantics to more-practical systems as well.

A few other works provide mechanized proofs of noninterference using different techniques [16, 3, 52]. However, each verifies existing paper proofs [52] or mechanizes an existing proof technique designed for a single-language setting [16, 3] (e.g., parametricity [3] or logical relations [16]). This work is unique among mechanizations of noninterference in its use denotational semantics designed to support multi-language settings.

Originally defined by Xia et al. [57], ITrees are based on free monads and their derivatives [22, 23, 50]. This gives rise to a natural interpretation of effects via monad transformers [19, 26] that behave like algebraic-effect handlers [47, 10, 38, 37, 35, 34]. The information-flow community also studies effects deeply since they can leak information. Traditionally, information-flow languages use a program-counter label to reason about effects, as we saw in Section 5. Recent work by Hirsch and Cecchetti [17] connects program-counter labels with monads, giving the former semantics using the latter.

Secure compilation is a very active research area. For instance, Barthe et al. [8] show how to securely compile to a low-level ASM-like target language. However, they use a type system for the target language to enforce security. Other efforts focus on particular language features, such as cryptographic constant time [9]. Moreover, until recently, most work on secure compilation focused on fully-abstract compilation [25]. Unfortunately, Abate et al. [2] recently showed that full abstraction is not sufficient to guarantee preservation of hyperproperties like noninterference. Our Mixed Transitivity theorems (Theorems 8 and 15) show that *equivalence-preserving* compilation does preserve noninterference.

Beyond work on secure compilation, most work on noninterference does not address multiple interacting languages. In one notable exception, Focardi et al. [14] examine the relationship between a process-calculus-based notion of security and simple imperative language with information-flow control, similar to IMP. They translate their version of IMP into CCS and show that they preserve IMP’s security guarantees. However, their work contains only pencil-and-paper proofs, rather than formally verifying their translation or its security.

Finally, this work focuses on an approach for verifying language toolchains, but running any program requires hardware. Most language-based security and verification work assumes the hardware is predictable and reliable, but cannot enforce security. Hardware enforcement of information-security properties [63, 58] provides dynamic enforcement of properties like noninterference at the cost of space and power usage. Combining these mechanisms with our approach could reduce the overhead of hardware enforcement for verified-secure programs and provide a means to guarantee that interactions with unverified programs remain safe.

8 Conclusion

This paper uses ITrees to reason semantically about noninterference. Our main technical contributions are two new indistinguishability relations on ITrees that we use to define noninterference – one progress sensitive and one progress insensitive – and their metatheory. While both noninterference definitions are coinductive, our metatheory library supports verifying properties of a language toolchain with no direct use of coinduction.

The two indistinguishability relations describe security in many settings, and we plan to include them in the ITrees library. Importantly, because they do not place any restrictions on the events in an ITree, they can be used for reasoning about a variety of language features. However, we recognize that many variations of noninterference appear in the literature, depending on the adversarial model and desired language features. For instance, *declassification* allows private information to be made public in controlled circumstances, creating a need for more complicated security conditions. We hope that the relations studied here both become the basis of verification efforts larger than our case study *and* that they serve as a starting point for further exploration of indistinguishability relations for ITrees.

References

- 1 Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1999. doi:10.1145/292540.292555.
- 2 Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hrițcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *IEEE Computer Security Foundations Symposium (CSF)*, 2019. doi:10.1109/CSF.2019.00025.
- 3 Maximilian Alghed and Jean-Philippe Bernardy. Simple noninterference from parametricity. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341693.
- 4 Maximilian Alghed and Alejandro Russo. Encoding dcc in haskell. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017. doi:10.1145/3139337.3139338.
- 5 Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *IEEE Computer Security Foundations Symposium (CSF)*, July 2015. doi:10.1109/CSF.2015.42.
- 6 Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *European Symposium on Research in Computer Security (ESORICS)*, pages 333–348. Springer, 2008. doi:10.1007/978-3-540-88313-5_22.
- 7 Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hrițcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *SIGPLAN Not.*, 49(1):165–178, January 2014. doi:10.1145/2578855.2535839.
- 8 Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 2–15, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- 9 Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic “constant time”. In *IEEE Computer Security Foundations Symposium (CSF)*, 2018. doi:10.1109/CSF.2018.00031.
- 10 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, January 2015.
- 11 Johan Bay and Aslan Askarov. Reconciling progress-insensitive noninterference and declassification. In *IEEE Computer Security Foundations Symposium (CSF)*, June 2020. doi:10.1109/CSF49147.2020.00015.
- 12 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security (JCS)*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.
- 13 Riccardo Focardi, Carla Piazza, and Sabina Rossi. Proof methods for bisimulation based information flow security. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2002.
- 14 Riccardo Focardi, Sabrina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *FoSSaCS*, 2005. doi:10.1007/978-3-540-31982-5_19.
- 15 Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy (S&P)*, 1982. doi:10.1109/SP.1982.10014.

- 16 Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434291.
- 17 Andrew K. Hirsch and Ethan Cecchetti. Giving semantics to program-counter labels via secure effects. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021. doi:10.1145/3434316.
- 18 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013. doi:10.1145/2429069.2429093.
- 19 Mauro Jaskelioff. Modular monad transformers. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 64–79, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 20 Limin Jia and Steve Zdancewic. Encoding information flow in Aura. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 17–29, 2009.
- 21 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2015. doi:10.1145/2676726.2676980.
- 22 Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105, 2015. doi:10.1145/2804302.2804319.
- 23 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. *ACM SIGPLAN Notices*, 48(12):59–70, 2013.
- 24 Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From c to interaction trees: Specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2019. doi:10.1145/3293880.3294106.
- 25 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.
- 26 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1995. doi:10.1145/199448.199528.
- 27 Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security (JCS)*, 25(4–5):319–321, May 2017. doi:10.3233/JCS-0559.
- 28 Tom Magrino, Jed Liu, Owen Arden, Chin Isradisaikul, and Andrew C. Myers. Jif 3.5: Java information flow. Software release, 2016. URL: <https://www.cs.cornell.edu/jif>.
- 29 Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2018. Version 8.8.1. URL: <http://coq.inria.fr>.
- 30 Mae P. Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018. doi:10.1145/3192366.3192375.
- 31 Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23, June 1989. Full version, titled *Notions of Computation and Monads*, in *Information and Computation*, 93(1), pp. 55–92, 1991.
- 32 Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1999. doi:10.1145/292540.292561.
- 33 Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy (S&P)*, 1998. doi:10.1109/SECPRI.1998.674834.

- 34 Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- 35 Gordon Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 342–356, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- 36 Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.
- 37 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.
- 38 Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, 9(4), December 2013. doi:10.2168/LMCS-9(4:23)2013.
- 39 Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid information flow control. *Proceedings of the ACM on Programming Languages*, 4(ICFP), August 2020. doi:10.1145/3408987.
- 40 François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, January 2003. doi:10.1145/596980.596983.
- 41 Willard Rafnsson and Andrei Sabelfeld. Compositional information-flow security for interactive systems. In *IEEE Computer Security Foundations Symposium (CSF)*, 2014. doi:10.1109/CSF.2013.8.
- 42 Vineet Rajani and Deepak Garg. Types for information flow control: Labeling granularity and semantic models. In *IEEE Computer Security Foundations Symposium (CSF)*, 2018. doi:10.1109/CSF.2018.00024.
- 43 John Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 1983.
- 44 Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *ACM SIGPLAN Haskell Symposium*, 2008. doi:10.1145/1411286.1411289.
- 45 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. doi:10.1109/JSAC.2002.806121.
- 46 Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001. doi:10.1023/A:1011553200337.
- 47 Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskieloff. Monad transformers and algebraic effects: What binds them together. Technical Report CW699, Department of Computer Science, KU Leuven, 2016.
- 48 Geoffery Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Computer Security Foundations Workshop (CSFW)*, 2003. doi:10.1109/CSFW.2003.1212701.
- 49 Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Nordic Conference on Security IT Systems (NordSec)*, October 2011. doi:10.1007/978-3-642-29615-4_16.
- 50 Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008. doi:10.1017/S0956796808006758.
- 51 Tsa-ching Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *IEEE Computer Security Foundations Symposium (CSF)*, 2007. doi:10.1109/CSF.2007.6.
- 52 Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS '16*, pages 15–28, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2993600.2993608.

- 53 Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. MAC: A verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming (JLAMP)*, 95, 2018. doi:10.1016/j.jlamp.2017.12.003.
- 54 Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. From fine-to coarse-grained dynamic information flow control and back. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019. doi:10.1145/3290389.
- 55 Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 1997. doi:10.1109/CSFW.1997.596807.
- 56 Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3), 1996. doi:10.3233/JCS-1996-42-304.
- 57 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL), January 2020. doi:10.1145/3371119.
- 58 Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *IEEE Computer Security Foundations Symposium (CSF)*, June 2019. doi:10.1109/CSF.2019.00026.
- 59 Yannick Zakowski, Calvin Beck, Irene Yoon, Ilya Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 2021.
- 60 Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, January 2020.
- 61 Steve Zdancewic and Andrew C Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2-3), 2002. doi:10.1023/A:1020843229247.
- 62 Nikolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, November 2011. doi:10.1145/2018396.2018419.
- 63 Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2012. doi:10.1145/2254064.2254078.
- 64 Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2005. doi:10.1109/CSFW.2005.16.

Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification

Lucas Silver ✉

University of Pennsylvania, Philadelphia, PA, USA

Eddy Westbrook ✉

Galois, Inc., Portland, OR, USA

Matthew Yacavone ✉

Galois, Inc., Portland, OR, USA

Ryan Scott ✉

Galois, Inc., Portland, OR, USA

Abstract

This paper presents a specification framework for monadic, recursive, interactive programs that supports auto-active verification, an approach that combines user-provided guidance with automatic verification techniques. This verification tool is designed to have the flexibility of a manual approach to verification along with the usability benefits of automatic approaches. We accomplish this by augmenting Interaction Trees, a Coq datastructure for representing effectful computations, with logical quantifier events. We show that this yields a language of specifications that are easy to understand, automatable, and are powerful enough to handle properties that involve non-termination. Our framework is implemented as a library in Coq. We demonstrate the effectiveness of this framework by verifying real, low-level code.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Theory of computation → Programming logic; Theory of computation → Separation logic

Keywords and phrases coinduction, specification, verification, monads

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.30

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.8>

Software (Source Code): <https://github.com/GaloisInc/entree-specs>

archived at `swh:1:dir:167704c70f9a14ef7d70a06b5bac5eb44fdb71e1`

1 Introduction

Formal verification is starting to see adoption in industry as a tool for ensuring the security and correctness of software. For instance, the formally verified seL4 microkernel [13] has established a foundation that is seeing investment from a wide variety of industrial partners. Block-chain companies are using formal verification to ensure the security of cryptocurrency [15]. Amazon has even incorporated formal verification into the CI/CD process of their s2n cryptographic library [7].

Unfortunately, formal verification still remains expensive, not just in terms of time and effort but also in terms of the expertise required to formally verify a system. A number of powerful frameworks have been developed for manual formal verification, including Iris [12], VST [2], and FCSL [24]. These frameworks can specify a wide array of behaviors on a wide array of languages, but they require an expert to be used effectively. Other powerful frameworks have been developed for automatic verification, including approaches such as



© Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 30; pp. 30:1–30:26



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



bounded model-checking [4] and property-directed reachability [5]. While these approaches can be operated by non-experts, they are limited in their expressiveness, leaving important properties unverified.

It is particularly difficult to reason about low-level code that contains complicated manipulations of pointer structures on the heap, as is common in languages like C, C++, and LLVM. Recently, researchers have tackled this problem using the observation that programs that are well-typed in a memory-safe, Rust-like type system are basically functional programs [9, 17, 18, 3, 10]. That is, there exists a program in a functional language whose behavior is equivalent to the original, heap-manipulating program. We call this functional program a *functional specification*. While many projects rely only implicitly on the functional specification, some, like the Heapster project [9], reify functional specifications into concrete code. Engineers can then verify properties about the derived functional code, and ensure those properties hold on the original program.

The Heapster tool consists of two components: a memory-safe type system for LLVM code, and a translation tool that produces an equivalent functional program from any well-typed LLVM program. Heapster uses these components to break verification of heap manipulating programs into two phases: a memory-safe type-checking phase that generates a monadic, recursive, interactive program that is equivalent to the original program; and a behavior-verification phase that ensures that the generated program has the correct behavior. Previous work has left open major questions about the behavior verification phase, namely, what should the language of specifications be and how do we actually prove that the programs satisfy the specifications.

This work answers these questions by developing a logic well-suited to reasoning about the programs output by Heapster, as well as tools to work with these logical formulae. Taken together, the Heapster tool and this work form a two-step pipeline for verifying low-level, heap manipulating programs. Heapster transforms low-level, heap manipulating programs into equivalent functional programs. The techniques in this paper enable proof engineers to write and prove specifications over the resulting functional programs.

In this work, we present *interaction tree specifications*, or ITree specifications. ITree specifications are an *auto-active verification framework* for *monadic, recursive, interactive programs* based on *interaction trees* [29], or ITrees. Auto-active verification is a verification technique that merges user input and automated reasoning to leverage the benefits of each. Monadic, recursive, interactive programs have the ability to diverge, can interact with their environment, but otherwise act as pure functional programs. Interactions with the environment can include making a system call, sending a message from a server, and throwing an error. ITrees are a model for monadic, recursive, interactive programs formalized in Coq. ITree specifications are designed to be able to write and verify specifications about the output programs of the Heapster translation tool, which are written in terms of ITrees.

The main body of work that takes on the task of verifying monadic programs is the Dijkstra monad literature [16, 28, 1, 27]. However, most of the Dijkstra monad literature cannot handle the kinds of termination sensitive specifications that we need. These papers either assume a strongly normalizing language, or handle only partial specifications. The exception to this is the work of Silver and Zdancewic [25]. However, while that work does have a rich enough specification language for our goals, it has two significant shortcomings. First, the work provides no reasoning principles for arbitrary recursive specifications. Second, the work does not attempt to automate the verification of these specifications. Our work accomplishes both of these goals.

This work is based on the idea of augmenting ITrees with operations for logical quantifiers. We show that this idea leads to a language of specifications that is:

- easy to read, because the specifications are simply programs annotated with logical quantifiers,
- capable of encoding recursive specifications, because the underlying computational language has a powerful recursion operator, and
- amenable to auto-active verification, because specifications are syntactic constructs enabling syntax-directed inference rules.

ITrees represent computations as potentially infinite trees whose nodes are labelled with *events*. Events are syntactic representations of computational effects, like raising an error, or sending data from a server. ITrees can be used to represent the semantics of recursive, monadic, interactive programs. ITree specifications are ITrees enriched with events for logical quantifiers. This language of specifications has the capability to express purely executable computations, fully abstract specifications, and combinations of both. For example, consider the following executable specification `server_impl` for a simple server program that sorts lists which are sent to it:

```
Definition server_impl : unit → itree_spec E void :=
  rec_fix_spec (fun rec _ =>
    l ← trigger rcvE;;
    ls ← sort l;;
    trigger (sendE ls);;
    rec tt
  ).
```

This specification is defined with `rec_fix_spec`, a recursion operator (defined in Section 4) where applications of the `rec` argument correspond to recursive calls. The body of the recursive function first calls `trigger rcvE`, which triggers the use of the receive event `rcvE`, causing the program to wait to receive data. The list `l` that is received is then passed to the `sort` function, defined in Section 6, which is a recursive implementation of the merge sort algorithm. Finally, the sorted list returned by `sort` is sent as a response with `trigger (sendE ls)`, and the server program loops back to the beginning by calling `rec`.

Now, consider the following specification of the behavior of our server using a combination of executable and abstract features:

```
Definition server_spec : unit → itree_spec E void :=
  rec_fix_spec (fun rec _ =>
    l ← trigger rcvE;;
    ls ← exists_spec (list nat);;
    assert_spec (Permutation l ls);;
    assert_spec (sorted ls);;
    trigger (sendE ls);;
    rec tt).
```

This function acts mostly like `server_impl` but, instead of computing a sorted list, it uses the existential quantification operation `exists_spec` to introduce the list value `ls`, which it then asserts is a sorted permutation of the initial list. By leaving this part of the specification abstract, it allows the user to express that it is unimportant how the list is sorted, as long as the response is a sorted permutation of the input list. The send and receive events, however, are left concrete, allowing the user to specify what monadic events should be triggered in what order. This specification implicitly defines a liveness property of the server, it will reject any program that fails to eventually perform the next send or receive. By using a single language for programs and specifications, our approach provides a natural way for users to control how concrete or abstract the various portions of their specifications are. Our approach then provides auto-active tools for proving that programs refine these specifications.

30:4 Interaction Tree Specifications

```
Class EncodingType (E:Type) : Type :=  
  response_type : E → Type.
```

■ **Figure 1** EncodingType typeclass definition.

Necessary background explaining ITrees and Heapster is given in Section 2 and Section 3. The contributions of this paper are as follows:

- ITree specifications, a data structure for representing specifications over monadic, recursive, interactive programs, presented in Section 4
- a specification refinement relation over ITree specifications, along with collection of verified, syntax-directed proof rules for refinement also presented in Section 4,
- tools for encoding and proving refinements involving total correctness specifications in ITree specifications presented in Section 5,
- an auto-active verification technique briefly discussed in Section 6
- an evaluation of the presented techniques in the form of verifying a collection of realistic C functions using ITree specifications and Heapster presented in Section 6.

2 Background

ITrees are a formalization for denotational semantics implemented as a coinductive variant of the free monad in Coq. ITrees represent programs as potentially infinite trees. The nodes of these trees are labelled with *events*. Events can, depending on the context, either represent algebraic effects or recursive function calls. The ITree type is parameterized by a return type R and a type family E , where E has an instance of the `EncodingType` type class defined in Figure 1. The `EncodingType` type class consists of function, named `response_type`, from E to `Type`. A value of type `itree E R` is a potentially infinite tree whose internal nodes are each labelled with an *event* e of type E , with one branch for each element of the `response_type e` whose leaves are labelled with an element of type R . Such a tree represents an effectful computation, where the leaves represent termination of the computation with a return value in R while the nodes represent uses of monadic effects. The event e of type E that labels a node represents a monadic effect that returns a value of type `response_type e`, and the children of that node represent the possible continuations of that computation depending on the return value of the effect. This is formalized in the following Coq code¹.

```
CoInductive itree (E : Type) {EncodingType E} (R : Type) :=  
  | Ret (r : R)  
  | Tau (t : itree E R)  
  | Vis (e : E) (k : response_type e → itree E R).
```

The ITree datatype has three constructors. The `Ret` constructor represents a pure computation that simply returns a value. The `Ret` constructor forms the leaves of an ITree. The `Tau` constructor represents one step of silent internal computation followed by another ITree. Finally, the `Vis` constructor contains an event e along with a continuation function k which defines all the branches of this `Vis` node.

Because ITrees are defined coinductively, we can construct ITrees with infinitely long branches. Such ITrees represent divergent computations. For example, the following code describes an ITree that consists of an infinite stream of `Tau` constructors with no events.

¹ In the actual formalization, we use a negative coinductive types presentation of this data structure.


```

Class ReSum (E1 : Type) (E2 : Type) `{EncodingType E1} `{EncodingType E2} :=
{
  resum : E1 → E2;
  resum_ret : forall {e : E1}, response_type (resum e) → response_type e;
}.

```

```

Notation "E1 -< E2" := (ReSum E1 E2) (at level 10).

```

```

Definition trigger {E1 E2} `{EncodingType E1} `{EncodingType E2} `{E1 -< E2} :forall (e1
  : E1), (itree E2 (response_type e1)) :=
  fun e => Vis (resum e) (fun x => Ret (resum_ret x)).

```

■ **Figure 2** ReSum Definition.

```

CoFixpoint spin : itree E R := Tau spin.

```

In practice, ITrees often end up using an event type family E that is a composition of several smaller type families combined in a large sum. This can easily clutter and complicate the notation. To avoid this burden, the ITrees library introduces the `ReSum` typeclass defined in Figure 2. An instance of `ReSum E1 E2`, written `E1 -< E2`, contains two functions: the `resum` function that injects an element of $E1$ into $E2$, and the `resum_ret` function that maps elements from the response type of `resum e` to the response type of e . It can be thought of as a kind of subevent typeclass. The `ReSum` typeclass allows for the definition of the `trigger` function in Figure 2. The `trigger` function takes an event $e : E1$ and injects it into `itree E (response_type e)` by injecting e into $E2$, placing that in a `Vis` node, and applying the `resum_ret` function to the response.

2.1 Equivalence up to Tau

One of the major advantages of the ITrees library is its rich equational theory. The primary notion of equivalence used for ITrees is called `eutt` or *equivalence up to tau*. Xia et al. [29] defines `eutt` as a bisimulation relation that quotients out finite differences in the number of `Tau` constructors. We use this relation because `Tau` constructors are supposed to indicate *silent* steps of computation. Ignoring finite numbers of `Tau` constructors lets us equate two ITrees that vary only in the number of silent computation steps.

The `eutt` relation is parameterized by a relation RR over return values. If the relation RR is *heterogeneous*, relating values over distinct types $R1$ and $R2$, then `eutt RR` is also a heterogeneous relation over `itree E R1` and `itree E R2`. Intuitively, if `eutt RR t1 t2`, then the `Vis` nodes of $t1$ precisely match those of $t2$, and if equivalent paths in $t1$ and $t2$ lead to the leaves `Ret r1` and `Ret r2` then the values $r1$ and $r2$ are related by RR . Often, we are interested in `eutt eq` and denote this relation with the symbol \approx .

The `eutt` relation is implemented in Coq using both *inductive* and *coinductive* techniques. Observe the following definition of `eutt`:

```

Inductive euttF (RR : R1 → R2 → Prop) (sim : itree E R1 → itree E R2 → Prop) :
  itree E R1 → itree E R2 → Prop :=
| eutt_Ret (r1 : R1) (r2 : R2) : euttF RR sim (Ret r1) (Ret r2)
| eutt_Tau (t1 : itree E R1) (t2 : itree E R2) :
  sim t1 t2 → euttF RR sim (Tau t1) (Tau t2)
| eutt_Vis (e : E) (k1 : response_type e → itree E R1)
  (k2 : response_type e → itree E R2) :
  (forall a, sim (k1 a) (k2 a)) → euttF RR sim (Vis e k1) (Vis e k2)
| eutt_TauL (t1 : itree E R1) (t2 : itree E R2) :

```

30:6 Interaction Tree Specifications

Example `spin ≈ spin`.

Example `Tau (Ret 0) ≈ Ret 0`.

Example `~(spin ≈ Ret 0)`.

■ **Figure 3** `eutt` Examples.

```
euttF RR sim t1 t2 → euttF RR sim (Tau t1) t2
| eutt_TauR (t1 : itree E R1) (t2 : itree E R2) :
  euttF RR sim t1 t2 → euttF RR sim t1 (Tau t2).
```

Definition `eutt (RR : R1 → R2 → Prop) := gfp (euttF RR)`.

The `euttF` relation is an inductively defined relation, defined in terms of the `sim` argument. The `eutt` relation is then defined as the greatest fixpoint of `euttF`. In this paper, all greatest fixpoints are defined using the `paco` library[11] for coinductive proofs. Calls to the `sim` argument in the definition of `euttF` correspond to coinductive calls of `eutt`. Recursive calls to `euttF` correspond to inductive calls of `eutt`. This method of defining `eutt` allows the coinductive constructors to be called infinitely often in sequence, while only a finite number of calls to inductive constructors can be called without an intervening call to a coinductive constructor. Specifically, only finitely many `eutt_TauL` and `eutt_TauR` steps, that remove a `Tau` from only one side, are allowed before one of the remaining rules must be used to relate the same constructor on both sides.

This definition allows us to achieve our goal of ignoring any finite difference in numbers of `Tau` constructors. In particular the equations and inequalities presented in Figure 3 hold.

ITrees form a monad. Monads are type families with a `ret` combinator that denotes a pure value, and a `bind` combinator that sequentially composes two monadic computations into one. The `ret` combinator is implemented with the `Ret` constructor, while the `bind t k` combinator is implemented as a coinductive function that traverses the ITree `t` and replaces each leaf `Ret r` with the new subtree `k r`. This is implemented in the following Coq code:

```
CoFixpoint bind (t : itree E R) (k : R → itree E S) :=
  match t with
  | Ret r ⇒ k r
  | Tau t ⇒ Tau (bind t k)
  | Vis e kvis ⇒ Vis e (fun x ⇒ bind (kvis x) k)
  end.
```

2.2 Mutually Recursive Computations

This section explains the recursion operator introduced by Xia et al. [29]. That work demonstrated how to use events as a piece of syntax for writing collections of mutually recursive functions over ITrees. Specifically, it introduced the `mrec` combinator, which lifts a collection of function bodies that syntactically reference one another to a collection of actually recursive functions. A similar recursion combinator is used extensively in Section 4 and Section 6.

When using the `mrec` combinator, you must first choose an event type `D`, with an `EncodingType` instance, to serve as the type of recursive calls. An element `d : D` packages together the choice of the function being called along with the arguments being supplied to that function. The return type of the function call `d` is `response_type d`. In this context, an ITree with the type `itree (D + E) R` represents the body of a mutually recursive function viewing the recursive calls as inert `D` events. This ITree defines a recursive function in terms of

```

Variant evenoddE : Type:=
  | even (n : nat) : evenoddE
  | odd (n : nat) : evenoddE.
Instance EncodingType_evenoddE : EncodingType evenoddE := fun _ => bool.

Definition evenodd_body : forall eo : evenoddE, (itree (evenoddE + voidE)) (
  response_type eo) :=
  fun eo =>
    match eo with
    | even n => if Nat.eqb n 0
                then Ret true
                else trigger (odd (n -1))
    | odd n => if Nat.eqb n 0
                then Ret false
                else trigger (even (n -1))
    end.
Definition evenodd : evenoddE → itree voidE bool :=
  mrec evenodd_body.

```

■ **Figure 4** evenodd Definition.

syntactic recursive calls. In order to resolve these syntactic recursive calls, we need a mapping from recursive calls to a single layer of unfolding of the recursive function. This is represented as a function of type `bodies : forall (d:D), itree (D + E) (response_type d)`. The variable name `bodies` refers to the fact that this term represents the body of each function in this collection of mutually recursive functions. We can then take this ITree, corecursively replace each `d : D` event with the unfolded function body `bodies d`, and then repeat the process with the resulting ITree. This is formalized in the following `interp_mrec` function.

```

CoFixpoint interp_mrec {R : Type}
  (bodies : forall (d:D), itree (D + E) (response_type d))
  (t : itree (D + E) R) : itree E R :=
  match t with
  | Ret r => Ret r
  | Tau t => Tau (interp_mrec bodies t)
  | Vis (inr e) k => Vis e (fun x => interp_mrec bodies (k x))
  | Vis (inl d) k => Tau (interp_mrec bodies (bind (bodies d) k))
  end.

```

Given this function that can resolve the recursive calls in an ITree, we can define the `mrec` function that takes an initial recursive call `init : D` and computes its result.

```

Definition mrec (bodies : forall (d:D), itree (D + E) (response_type d)) (init : D)
  :=
  interp_mrec bodies (bodies init).

```

Figure 4 provides an example of a mutually recursive function defined with `mrec`. The `evenoddE` type represents calls to compute the parity of a natural number. The `evenodd` function computes either the `even` or the `odd` function depending on the initial recursive call event that it is given. The `evenodd` function defines these computations mutually recursively using the `mrec` function.

This section briefly introduces the classes of relations that we will need in order to reason about specification refinement in the presence of mutually recursive computations. The definition of `eutt` is parameterized by a return relation, making it easy to define a relation for ITrees that have identical tree structures up to Taus, with identical event nodes, but allows freedom to choose what conditions to enforce on return values. It is natural to consider generalizing `eutt` to allow variation not only in the return values but also in the event nodes.

```

Definition Rel (A B : Type) : Type := A → B → Prop.
Definition PostRel (D1 D2 : Type) {EncodingType D1} {EncodingType D2} : Type :=
  forall (d1 : D1) (d2 : D2), response_type d1 → response_type d2 → Prop.

Inductive RComposePostRel
  (R1 : Rel D1 D2) (R2 : Rel D2 D3) (PR1 : PostRel D1 D2) (PR2 : PostRel D2 D3) :
  PostRel D1 D3 :=
  | RComposePostRel_intros (d1 : D1) (d3 : D3) (a : response_type d1) (c :
    response_type d3) :
    (forall (d2 : D2), R1 d1 d2 → R2 d2 d3 →
      exists b, PR1 d1 d2 a b ∧ PR2 d2 d3 b c) →
    RComposePostRel R1 R2 PR1 PR2 d1 d3 a c.

```

■ **Figure 5** Heterogeneous Event Relation Types.

This kind of generalization is explored in Silver and Zdancewic [25]². The generalized relation analyzes uninterpreted events, typically those representing recursive function calls, with respect to pre-conditions and post-conditions. We want to relate `Vis` nodes whose events satisfy the pre-condition and whose continuations are related given any inputs that satisfy the post-condition. This corresponds to assuming that two function calls return related outputs as long as they are given related inputs.

Definitions of pre-condition and post-condition types are presented in Figure 5. Pre-conditions, `Rel`, are encoded as two-argument, heterogeneous relations, i.e. functions of type $D \rightarrow E \rightarrow \text{Prop}$, and utilize standard relational combinators like relational sums, `sum_rel`, and relational composition, `rcompose`. Post-conditions, `PostRel`, are encoded as four-argument, dependent relations. In particular, `forall (d:D) (e:E), encoded_by d → encoded_by e → Prop`, where both `D` and `E` have an `EncodingType` instance. Intuitively, post-conditions are a function from events to relations over their response types. These post-conditions admit a standard definition of relational sums. For relational composition, in addition to requiring two `PostRel` relations, it also requires two standard relations, called *coordinating relations*. The full definition is presented in Figure 5.

To relate four values `d1:D1`, `d3:D3`, `a:encoded_by d1`, `c:encoded_by d3`, we require that given any `d2:D2` that is related by the coordinating relations to `d1` and `d3`, there exists a `b:encoded_by d2` such that both `PR3 d1 d2 a b` and `PR4 d2 d3 b c`.

Later in the paper, we recover an `eutt`-like definition of specification refinement by specializing the event relations to be an appropriate form of equality. For `Rel`, this is precisely the equality relation. For `PostRel`, we define an inductive datatype that enforces equality on response values.

```

Variant PostRelEq : PostRel E E :=
  PostRelEq_intro e a : PostRelEq e e a a.

```

3 Specification Extraction with Heapster

This section introduces the Heapster tool for specification extraction. We present Heapster in order to provide context for the evaluation of this work in Section 6. In the evaluation, we demonstrate how effective ITree specifications can be when paired with a tool like Heapster. We start with a collection of low-level, heap manipulating C programs, use Heapster to produce equivalent functional programs, and finally use ITree specifications to specify and verify the output programs.

² In Silver and Zdancewic [25] this relation is referred to as `euttEv`. It has since been renamed to `ruTT` in release branches of the Interaction Trees library.

Value Types	T	$::=$	$\text{bv } n \mid \text{llvmptr } n \mid \dots$
Expressions	e	$::=$	$n \mid \text{llvmword } e \mid \dots$
RW Modality	rw	$::=$	$W \mid R$
Permissions	τ	$::=$	$\text{ptr}((rw, e) \mapsto \tau) \mid \tau_1 * \tau_2 \mid \tau_1 \vee \tau_2 \mid \exists x:T.\tau \mid \text{eq}(e) \mid \mu X.\tau \mid X \mid \dots$

■ **Figure 6** An Abbreviated Grammar of the Heapster Type System.

There is a growing body of work [9, 17, 18, 3] based on the idea that programs that satisfy memory-safe type systems like Rust can be represented with equivalent functional programs. Rust’s pointer discipline, which ensures that all pointers in a program are either shared read or exclusive write, allows us to reason about the effects of pointer updates purely locally. This locality property can be used to define a pure functional model, referred to as a *functional specification*, of the behaviors of a program, which can in turn be used to verify properties of that program.

Whereas some work uses this notion of a functional model implicitly, *specification extraction* is the idea that the functional model can be extracted automatically as an artifact that can be used for verification. Specification extraction separates verification into two phases: a type-checking phase, where the functions in a program are type-checked against user-specified memory-safe types; and a behavior verification phase, where the user verifies the specifications that are extracted from this type-checking process. The Heapster tool [9] is an implementation of the idea of specification extraction. Heapster provides a memory-safe, Rust-like type system for LLVM, along with a typechecker. Heapster also provides a translation from well-typed LLVM programs to monadic, recursive, interactive programs, modeled with ITrees, that describe a behavioral model of the original program. This translation is inspired by the Curry-Howard isomorphism. Heapster types are essentially a form of logical propositions regarding the heap, so, by the Curry-Howard isomorphism, it is natural to view typing derivations, a form of proof, as a program. We give a brief overview of the Heapster type system and its specification extraction process in this section and illustrate it with an example.

The Heapster type system is a permission type system. Typing assertions of the form $x : \tau$ mean that the current function holds permissions to perform actions allowed by τ on the value contained in variable x . The central permission construct of Heapster is the permission to read or write a pointer value. Like Rust, Heapster is an affine type system, meaning that the permissions held by a function can change at different points in the function. In particular, a command can consume a permission, preventing further commands from using that permission again. Also like Rust, Heapster allows read-only permissions to be duplicated, allowing multiple read-only pointers to the same address, but does not allow write permissions to be duplicated. This enforces the invariant that all pointers are either shared read or exclusive write, a powerful property for proving memory-safety.

Figure 6 gives an abbreviated grammar for the Heapster type system. The value types T are inhabited by pieces of first order data. In particular, they contain the type $\text{bv } n$ of n -bit bitvectors (i.e., n -bit binary values) and the type $\text{llvmptr } n$ of n -bit LLVM values, among other value types not discussed here. Heapster uses the CompCert memory model [14], where LLVM values are either a word value or a pointer value represented as a pair of a memory region plus an offset in that region. The expressions e include numeric literals n and applications of the llvmword constructor of the LLVM value type to build an LLVM value from a word value.

30:10 Interaction Tree Specifications

The first permission type in Figure 6, $\text{ptr}(\langle rw, e \rangle \mapsto \tau)$, represents a permission to read or write (depending on rw) a pointer at offset e . Write permission always includes read permission. This permission also gives permission τ to whatever value is currently pointed to by the pointer with this permission. Permission type $\tau_1 * \tau_2$ is the separating conjunction of τ_1 and τ_2 , giving all of the permissions granted by τ_1 or τ_2 , where τ_1 and τ_2 contain no overlapping permissions. Permission type $\tau_1 \vee \tau_2$ is the disjunction of τ_1 and τ_2 , which either grants permissions τ_1 or τ_2 . The existential permission $\exists x:T.\tau$ gives permission τ for some value x of value type T . The equality permission $\text{eq}(e)$ states that a value is known to be equal to an expression e . This can be viewed as a permission to assume the given value equals e . Finally, $\mu X.\tau$ is the least fixed-point permission, where permission variable X is bound in τ . This satisfies the fixed-point property, that $\mu X.\tau$ is equivalent to $[\mu X.\tau/X]\tau$.

As a simple example, the user can define the Heapster type

```
int64 =  $\exists x:\text{bv } 64.\text{eq}(\text{llvmword } x)$ 
```

This Heapster type describes an LLVM word value, i.e., an LLVM value that equals `llvmword x` for some bitvector x .

As a slightly more involved example, consider the following definition of a linked list structure in C:

```
typedef struct list64_t { int64_t data;
                        struct list64_t *next; } list64_t;
```

A C value of type `list64_t*` represents a list, where a NULL pointer represents the empty list and a non-NULL pointer to a `list64_t` struct represents a list whose head is the 64-integer contained in the `data` field and whose tail is given by the `next` field.

The following Heapster type describes this linked list structure:

```
list64 $\langle rw \rangle$  =  $\mu X.\text{eq}(\text{llvmword } 0) \vee (\text{ptr}(\langle rw, 0 \rangle \mapsto \text{int64}) * \text{ptr}(\langle rw, 8 \rangle \mapsto X))$ 
```

The `list64 $\langle rw \rangle$` type is parameterized by a read-write modality rw , which says whether it describes a read-only or read-write pointer to a linked list. The permission states that the value it applies to either equals the NULL pointer, represented as `llvmword 0`, or points at offset 0 to a 64-bit integer and at offset 8^3 to an LLVM value that itself recursively satisfies the `list64 $\langle rw \rangle$` permission. Note that the fact that it is a least fixed-point implicitly requires the list to be loop-free.

Figure 7 illustrates the process of Heapster type-checking on a simple function `is_elem` that checks if 64-bit integer `x` is in the linked list `l`. Note that Heapster in fact operates on the LLVM code that results from compiling this C code, but the type-checking is easier to visualize on the C code rather than looking at its corresponding LLVM. Ignoring the Heapster types for the moment, which are displayed with a grey background in the figure, `is_elem` first checks if `l` is NULL, and if so returns 0 to indicate that the check has failed. If not, it checks if the head of the list in `l->data` equals `x`, and if so, returns 1. Otherwise, it recurses on the tail `l->next`.

The Heapster permissions for this function are

```
 $x:\text{int64}, l:\text{list64}\langle R \rangle \multimap r:\text{int64}$ 
```

³ We assume a 64-bit architecture, so offset 8 references the second value of a C struct.

```

int64_t is_elem (int64_t x, list64_t *l) {
  x:int64,l:list64(R)
  x:int64,l:eq(llvword 0) OR x:int64,l:ptr((R,0) ↦ int64) * ptr((R,8) ↦ list64(R))
  if (l == NULL) {
    x:int64,l:eq(llvword 0)
    return 0;
  } else {
    x:int64,l:ptr((R,0) ↦ int64) * ptr((R,8) ↦ list64(R))
    if (l->data == x) { return 1; }
    else {
      list64_t *l2 = l->next;
      x:int64,l:ptr((R,0) ↦ int64) * ptr((R,8) ↦ eq(12)),l2:list64(R)
      return is_elem (x, l2);
    }
  }
}

```

■ **Figure 7** Type-checking the `is_elem` Function Against Type $x:\text{int64}, l:\text{list64}(R) \multimap r:\text{int64}$.

The lollipop symbol, \multimap , is used to write Heapster function types. This type means that input x is a 64-bit integer and l is a read-only linked list pointer and the return value r is a 64-bit integer value.

To type-check `is_elem`, Heapster starts by assuming the input types for the arguments. This is displayed in the first grey box of Figure 7. In order to type-check the `NULL` comparison on `l`, Heapster must first unfold the recursive permission on `l` and then eliminate the resulting disjunctive permission. This latter step results in Heapster type-checking the remaining code twice, once for each branch of the disjunct. More specifically, the remaining code is type-checked once under the assumption that `l` equals `NULL` and once under the assumption that it points to a valid `list64_t` struct. In the first case, the `NULL` check is guaranteed to succeed, and so the `if` branch is taken with those permissions, while in the second, the `NULL` check is guaranteed to fail, so the `else` branch is taken.

In the `if` branch, the value `0` is returned. Heapster determines that this value satisfies the required output permission `int64`. In the `else` branch, `l->data` is read, by dereferencing `l` at offset `0`. This is allowed by the permissions on `l` at this point in the code. If the resulting value equals `x`, then `1` is returned, which also satisfies the output permission `int64`. Otherwise, `l->next` is read, by dereferencing `l` at offset `0`, and the result is assigned to local variable `l2`. This assigns `list64(R)` permission to `l2`. The permission on offset `8` of `l` is updated to indicate that the value currently stored there equals `12`. The `list64(R)` permission on `l2` is then used to type-check the subsequent recursive call to `is_elem`.

Once a function is type-checked, Heapster performs specification extraction to extract a pure functional specification of the function’s behavior. Specification extraction translates permission types to Coq types and typing derivations to Coq programs. The type translation is defined as follows:

$$\begin{array}{ll}
\llbracket \text{ptr}((rw, e) \mapsto \tau) \rrbracket &= \llbracket \tau \rrbracket & \llbracket \tau_1 * \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket * \llbracket \tau_2 \rrbracket \\
\llbracket \tau_1 \vee \tau_2 \rrbracket &= \llbracket \tau_1 \rrbracket + \llbracket \tau_2 \rrbracket & \llbracket \exists x:T. \tau \rrbracket &= \{x : \llbracket T \rrbracket \} \& \llbracket \tau \rrbracket \\
\llbracket \text{eq}(e) \rrbracket &= \text{unit} & \llbracket \mu X. \tau \rrbracket &= \text{user-specified type } A \\
&&&& \text{isomorphic to } \llbracket [\mu X. \tau / X] \tau \rrbracket
\end{array}$$

Pointer permissions $\text{ptr}((rw, e) \mapsto \tau)$ are translated to the result of translating the permission τ of the value that is pointed to. This means that specification extraction erases pointer types, which are no longer needed in the resulting functional code. Conjunctive permissions are

30:12 Interaction Tree Specifications

```

Definition is_elem_spec : bitvector 64 * list (bitvector 64) →
  itree_spec E (bitvector 64) :=
  rec_fix_spec (fun rec '(x,l) ⇒
    either
      unit (bitvector 64 * list (bitvector 64)) (* input types *)
      (itree_spec _ (bitvector 64))           (* output type *)
      (fun _ ⇒ Ret (intToBv 64 0))           (* nil case *)
      (fun '(hd,tl) ⇒
        if bvEq 64 hd x then Ret (intToBv 64 1) (* return 1 *)
        else rec (x,tl))                       (* recursive call *)
      (unfoldList 1)).                          (* unfolded argument *)

```

■ **Figure 8** Extracted Specification for `is_elem`.

translated to pairs, disjunctive permissions are translated to sums, and existential permissions are translated to dependent pairs (using a straightforward translation $\llbracket T \rrbracket$ of value types that we omit here). The equality type `eq(e)` is translated to the Coq unit type `unit`, meaning that they contain no data in the extracted specifications. We already proved the equality in the typechecking phase, and we have no use for the particular equality proof the typechecker provided. To translate a least fixed-point type $\mu X.\tau$, the user specifies a type that satisfies the fixed-point equation, meaning a pair of functions

$$\text{fold} : \llbracket [\mu X.\tau/X]\tau \rrbracket \rightarrow \llbracket \mu X.\tau \rrbracket \quad \text{unfold} : \llbracket \mu X.\tau \rrbracket \rightarrow \llbracket [\mu X.\tau/X]\tau \rrbracket$$

that form an isomorphism.

As an example, the translation of `int64` is the Coq sigma type `{x:bitvector 64 & unit}`. Note that Heapster will in fact optimize away the unnecessary `unit` type, yielding the type `bitvector 64`. As a slightly more complex example, in order to translate the `list64⟨rw⟩` described above, the user must provide a type `T` that is isomorphic to the type

```
unit + (bitvector 64 * T)
```

The simplest choice for `T` is the type `list (bitvector 64)`. In this way, the imperative linked list data structure defined above in `C` is translated to the pure functional list type.

Rather than defining the translation of Heapster typing derivations into Coq programs here, we illustrate the high-level concepts with our example and refer the interested reader to He et al. [9] for more detail. The translation of `is_elem` is given as a Coq specification `is_elem_spec` in Figure 8. At the top level, this specification uses `rec_fix_spec` to define a recursive function to match the recursive definition of `is_elem`. This binds a local variable `rec` to be used for recursive calls to the specification.

To understand the rest of the specification, we step through the Heapster type-checking depicted in Figure 7. The first step of that type assignment unfolds the permission type `list64⟨W⟩` on `l`. The corresponding portion of the specification is the call to `unfoldList`, which unfolds the input list `l` to a sum of a unit or the head and tail of the list. The next step of the Heapster type-checking is to eliminate the resulting disjunctive permission on `l`. The corresponding portion of the specification is a call to the `either` sum elimination function. In the left-hand case of the disjunctive elimination, the `NULL` test of the `C` program succeeds, and `0` is returned. Similarly, in the Coq specification, the `nil` case returns the `0` bitvector value.

In the right-hand case of the disjunctive elimination of the Heapster type-checking, the `NULL` test fails, and so `l` is a valid pointer to a `C` struct with `data` and `next` fields. This is represented by the pattern-match on the `cons` case in the Coq specification, yielding variables `hd` and `tl` for the head and tail of the list. The body of this case then tests whether the head

equals the input variable x , corresponding to the $x==1 \rightarrow \text{data}$ expression in the C program. If so, then the bitvector value 1 is returned. Otherwise, the specification performs a recursive call, passing the same value for x and the tail of the input list for 1.

4 ITree Specifications and Refinement

In this paper, we introduce a specialization of the `ITree` data type that encodes specifications over `ITrees`. To do this, we take some base event type family E , and extend it with constructors for universal and existential quantification. This is formalized in the following definition for `SpecEvent`.

```
Inductive SpecEvent (E : Type) {EncodingType E} : Type :=
| Spec_vis (e : E) : SpecEvent E
| Spec_forall (A : type) : SpecEvent E
| Spec_exists (A : type) : SpecEvent E
```

The `Spec_vis` constructor allows you to embed a base event $e : E$ into the type `SpecEvent E`. The `Spec_forall` constructor signifies universal quantification, and the `Spec_exists` constructor signifies existential quantification. For the purposes of specifying Heapster programs, we only need to quantify over a fixed grammar of first order types⁴. This includes natural numbers, bit vectors, functions, products, logical propositions, and sums. We have omitted the definition of the particular fixed grammar of types used in this work for space.

We define *ITree specifications* as the type of `ITrees` with a `SpecEvent` as the event type.

```
Definition itree_spec (E : Type) {EncodingType E} (R : Type) :=
  itree (SpecEvent E) R.
```

Because `ITree` specifications are actually a special kind of `ITree`, they inherit all the useful metatheory and code defined for `ITrees`. In particular, we can reason about them equationally with `eut`, and apply the monad functions to them.

4.1 ITree Specification Refinement

The notion that a program adheres to a specification is defined with the notion of refinement. Refinement is the main judgment involved in using `ITree` specifications, and is for instance the primary form of proof goal proved by the provided automation tool. Intuitively, the logical quantifier events mean that an `ITree` specification represents a set of computations. A fully concrete `ITree` specification, with no logical quantifier events, represents a singleton set, while a more abstract specification might represent a larger set. The refinement relation is then defined such that, if one `ITree` specification refines another, then the former represents a subset of the latter. So, for instance, if we prove that a concrete specification refines a more abstract specification, then we have shown that the singleton program in the set represented by the concrete specification satisfies the specification. Note that refinement is actually a coarser relation than subset; this is discussed later in Section 4.4.

The `ITree` specification refinement relation is based on the idea of refinement of logical formulae with the `eut` relation. As in a sequent calculus, we can eliminate quantifiers in our specification logic using quantifiers in the base logic, in this case `Coq`. Quantifiers on the right of a refinement get eliminated to the corresponding `Coq` quantifiers, while quantifiers on

⁴ While we could quantify over `Type` in these definitions, this introduces universe level constraints that we prefer to avoid

30:14 Interaction Tree Specifications

the left get eliminated to the dual of the corresponding Coq quantifier. This means that both a `Spec_forall` on the right and a `Spec_exists` on the left get eliminated to a Coq `forall`. And both a `Spec_exists` on the right and a `Spec_forall` on the left get eliminated to a Coq `exists`. ITree specifications form a lattice with refinement serving as the preorder, `Spec_forall` acting as the complete meet, and `Spec_exists` acting as the complete join. The portions of ITree specifications with computational content, including the `Ret` leaves, `Spec_vis` nodes, and silent `Tau` nodes, get compared as they do in the `eutt` relation.

The ITree specification refinement relation shares many mechanical details with the `eutt` relation. Both are defined by taking the greatest fixed point of an inductively defined relation to get a mixture of inductive and coinductive properties. Both behave identically on `Tau` and `Ret` nodes. The refinement relation differs in its inductive rules for eliminating logical quantifiers, and in its usage of heterogeneous event relations to enforce pre- and post-conditions on `Spec_vis` events. These pre- and post-conditions are necessary in order to give the refinement relation the flexibility needed to state the reasoning principle for `mrec`. The initial inductively defined relation, `refinesF`, contains the following header code.

```

Inductive refinesF
  (RPre : Rel E1 E2) (RPost : PostRel E1 E2) (RR : Rel R1 R2)
  (sim : itree_spec E1 R1 → itree_spec E2 R2 → Prop)
  : itree_spec E1 R1 → itree_spec E2 R2 → Prop :=

```

Much like in the definition of `euttF`, the `sim` argument represents corecursive calls of the `refines` relation, and the `RR` argument is the relation used for return. Unlike in `euttF`, `refinesF` takes in arguments for a `PreRel` and a `PostRel`. These arguments are included in order to represent pre- and post-conditions on mutually recursive function bodies.

The `refinesF` relation has several constructors that work precisely the same as the corresponding `euttF` constructors. These constructors define the relation's behavior on `Ret` and `Tau` nodes.

```

| refines_Ret (r1 : R1) (r2 : R2) : RR r1 r2 → refinesF RPre RPost RR sim (Ret r1)
  (Ret r2)
| refines_Tau (phi1 : itree_spec E1 R1) (phi2 : itree_spec E2 R2) : sim phi1 phi2
  →
  refinesF RPre RPost RR sim (Tau phi1) (Tau phi2)
| refines_TauL (t1 : itree_spec E1 R1) (t2 : itree_spec E2 R2) :
  refinesF RPre RPost RR sim t1 t2 → refinesF RPre RPost RR sim (Tau t1) t2
| refines_TauR (t1 : itree_spec E1 R1) (t2 : itree_spec E2 R2) :
  refinesF RPre RPost RR sim t1 t2 → refinesF RPre RPost RR sim t1 (Tau t2)

```

The constructor dealing with `Spec_vis` nodes generalizes the constructor dealing with `Vis` nodes in `euttF`. This constructor relates `Spec_vis` nodes as long as two conditions hold on the events, `e1` and `e2`, and the continuations, `k1` and `k2`. The ITree specifications must satisfy the precondition, by having `e1` and `e2` satisfy `RPre`. And the ITree specifications must satisfy the post condition by having `k1` a refine `k2` b, whenever `a` and `b` are related by `RPost e1 e2`.

```

| refines_Spec_vis (e1 : E1) (e2 : E2)
  (k1 : response_type e1 → itree_spec E1 R1) (k2 : response_type e2
  → itree_spec E2 R2) :
  RPre e1 e2 → (forall a b, RPost e1 e2 a b → sim (k1 a) (k2 b)) →
  refinesF RPre RPost RR sim (Vis (Spec_vis e1) k1) (Vis (Spec_vis e2) k2)

```

The added complications of this rule allow us to reason about mutually recursive functions. It ensures that related function outputs assume that function calls with arguments related by the precondition return values related by the post condition when analyzing mutually recursive functions.

Finally, we need constructors dealing with quantifier events. This definition uses only inductive constructors to eliminate quantifier events. We made this choice to avoid certain peculiar issues related to ITree specifications that consist of infinite trees of only quantifiers. Given coinductive constructors for quantifier events, we would be able to prove that such

```

Class CoveredType (A : Type) := {
  encoding : type;   surjection : response_type encoding → A;
  surjection_correct : forall a : A, exists x, surjection x = a; }.

Definition forall_spec {E}
  `{EncodingType E}
  (A:Type) `{CoveredType A} :
  itree_spec E A :=
  Vis (Spec_forall encoding)
  (fun x ⇒ Ret (surjection x)).

Definition exists_spec {E}
  `{EncodingType E}
  (A:Type) `{CoveredType A} :
  itree_spec E A :=
  Vis (Spec_exists encoding)
  (fun x ⇒ Ret (surjection x)).

Definition assume_spec {E}
  `{EncodingType E} (P : Prop) :
  itree_spec E unit :=
  forall_spec P;; Ret tt.

Definition assert_spec {E}
  `{EncodingType E} (P : Prop) :
  itree_spec E unit :=
  exists_spec P;; Ret tt.

```

■ **Figure 9** Basic Specifications.

ITree specifications both refine and are refined by any other arbitrary ITree specification. That choice would cause certain ITree specifications to serve as both the top and bottom elements of the refinement order. This would serve as a counterexample to the transitivity of refinement, a desired property. So we chose to only use inductive constructors for quantifier events. This means that ITree specifications that consist of infinite trees of only quantifiers cannot be related by refinement to any other ITree specifications.

Quantifiers on the right get directly translated into Coq level quantifiers.

```

| refines_forallR (t : itree_spec E1 R1) (A:type) (k : response_type A →
  itree_spec E2 R2) :
  (forall a, refinesF RPre RPost RR sim t (k a)) →
  refinesF RPre RPost RR sim t (Vis (Spec_forall A) k)
| refines_existsR (t : itree_spec E1 R1) (A : type) (k : response_type A →
  itree_spec E2 R2) :
  (exists a, refinesF RPre RPost RR sim t (k a)) →
  refinesF RPre RPost RR sim t (Vis (Spec_exists A) k)

```

Quantifiers on the left get translated into their dual quantifier at the Coq level. Eliminating a `Spec_forall` on the left gives you an `exists`. Eliminating a `Spec_exists` on the left gives you an `forall`.

```

| refines_forallL (A : type) (k : response_type (Spec_forall A) → itree_spec E1 R1)
  (t : itree_spec E2 R2) :
  (exists a, refinesF RPre RPost RR sim (k a) t) →
  refinesF RPre RPost RR sim (Vis (Spec_forall A) k) t
| refines_existsL (A : type) (k : response_type (Spec_exists A) → itree_spec E1 R1)
  (t : itree_spec E2 R2) :
  (forall a, refinesF RPre RPost RR sim (k a) t) →
  refinesF RPre RPost RR sim (Vis (Spec_exists A) k) t

```

This `refinesF` relation is used to define the `refines` relation as follows.

```

Definition refines RPre RPost RR := gfp (refinesF RPre RPost RR).

```

4.2 Padded ITrees

Useful refinement relations should respect the `eutt` relation. When using ITrees as a denotational semantics, `eutt` is the basis of any program equivalence relation. Equivalent programs and specifications should not be observationally different according to the refinement relation. However, the `refines` relation does not respect `eutt`

We can easily demonstrate this with the following three ITree specifications.

30:16 Interaction Tree Specifications

```
CoFixpoint spin : itree_spec E R := Tau spin.
CoFixpoint phi1 : itree_spec E R := Vis (Spec_forall t) (fun _ => Tau (phi1)).
CoFixpoint phi2 : itree_spec E R := Vis (Spec_forall t) (fun _ => phi2).
```

The `spin` specification represents a silently diverging computation. The `phi1` specification is an infinite stream that alternates between `Spec_forall` nodes and `Tau` constructors. The `phi2` specification is a similar `ITree` to `phi1` that just lacks the `Tau` nodes. As these `ITree` specifications all diverge along all paths and lack any `Spec_vis` nodes, the `RPre`, `RPost`, and `RR` relations that we choose do not matter. Given any choice for those relations, `spin` refines `phi1` as we can use the inductive `refines_forallL` rule to get rid of the `Spec_forall` nodes, allowing us to match `Tau` nodes on both trees and apply the coinductive `refines_Tau` rule. This process can be extended coinductively allowing us to construct the refinement proof. The `phi1` `ITree` specification is `eutt` to `phi2`, as the only difference between the specifications is a single `Tau` node after every `Vis_forall` node. However, `spin` does not refine `phi2`, as there is no coinductive constructor that we can apply in order to write a proof for these divergent `ITree` specifications. Problems like this arise with any `ITree` specifications that consist of infinitely many quantifier nodes with nothing between them.

To fix this problem, we restrict our focus to a subset of `ITrees` that does not include ones like `phi2`. This is the set of *padded* `ITrees`, in which every `Vis` node must be immediately followed by a `Tau`. We formalize this with the coinductive `padded` predicate, whose definition has been omitted to save space. The refinement relation does not distinguish between different `ITree` specifications that are `eutt` to one another as long as they are padded. This means that can rewrite one `ITree` specification into another under a refinement according to `eutt` as long as both are padded.

Furthermore, it is easy to take an arbitrary `ITree`, and turn it into a padded `ITree`. That is implemented by the `pad` function, which corecursively adds a `Tau` after every `Vis` node. From here, we can focus primarily on the following definition of `padded_refines` which pads out all `ITree` specifications before passing them to the `refines` relation.

```
Definition padded_refines RPre RPost RR phi1 phi2 :=
  refines RPre RPost RR (pad phi1) (pad phi2).
```

In Figure 9, we introduce several simple `ITree` specifications that implement quantification over some types, and assumption and assertion of propositions. The `forall_spec` and `exists_spec` specifications rely on the `CoveredType` type class. A `CoveredType` instance for a type `A` contains an element of the restricted type grammar, `encoding`, whose interpretation corresponds to `A`. It also contains a valid surjection from the interpreted type `response_type encoding` to the original type `A`. In practice, we always instantiate this surjection with the identity function, but this type class formalization gives us the tools that we need without needing to do too much dependently typed programming. We can use `forall_spec` and `exists_spec` to define assumption and assertion, respectively, as `Prop` is part of the restricted grammar of types that `SpecEvent` can quantify over.

4.3 Padded Refinement Meta Theory

This subsection introduces some of the useful, verified metatheory we provide for `ITree` specifications in terms of `padded_refines` relation.

We prove that we can compose refinement results with the monadic `bind` operator.

```
Theorem padded_refines_bind (phi1 : itree_spec E1 R1) (phi2 : itree_spec E2 R2)
  (kphi1 : R1 → itree_spec E1 S1)
  (kphi2 : R2 → itree_spec E2 S2) :
  padded_refines RPre RPost RR phi1 phi2 →
  (forall r1 r2, RR r1 r2 → padded_refines RPre RPost RS (kphi1 r1) (kphi2 r2)) →
  padded_refines RPre RPost RS (bind phi1 kphi1) (bind phi2 kphi2).
```

```

CoFixpoint interp_mrec_spec {R : Type}
  (bodies : forall (d:D), (itree_spec (D + E)) (response_type d)) (t : itree_spec (D + E)
    ) R) : itree_spec E R :=
  match t with
  | Ret r => Ret r
  | Tau t => Tau (interp_mrec_spec bodies t)
  | Vis (Spec_forall A) k => Vis (@Spec_forall E _ A) (fun x : response_type (Spec_forall
    A) => interp_mrec_spec bodies (k x))
  | Vis (Spec_exists A) k => Vis (@Spec_exists E _ A) (fun x => interp_mrec_spec bodies (
    k x))
  | Vis (Spec_vis (inr e)) k => Vis (Spec_vis e) (fun x => interp_mrec_spec bodies (k x))
  | Vis (Spec_vis (inl d)) k => Tau (interp_mrec_spec bodies (bind (bodies d) k))
  end.

```

```

Definition mrec_spec (bodies : forall (d:D), (itree_spec (D + E)) (response_type d)) (
  init : D) :=
  interp_mrec_spec bodies (bodies init).

```

■ **Figure 10** `mrec_spec` Definition.

We prove that the `padded_refines` relation is transitive. To state the transitivity result in full generality, we need to use the composition relation introduced in Figure 5.

```

Theorem padded_refines_trans : forall (phi1 : itree_spec E1 R1) (phi2 : itree_spec E2
  R2) (phi3 : itree_spec E3 R3),
  padded_refines RPre1 RPost1 RR1 phi1 phi2 →
  padded_refines RPre2 RPost2 RR2 phi2 phi3 →
  padded_refines (RCompose RPre1 RPre2)
    (RComposePostRel RPre1 RPre2 RPost1 RPost2) (RCompose RR1 RR2) phi1 phi3.

```

We prove a reasoning principle for mutually recursive specifications as well. To do this, we first provide a slightly different definition of mutual recursion that handles the quantifier events correctly, defined in Figure 10. The key to proving refinements between `mrec_spec` specifications is to use the `PreRel` and `PostRel` relations to establish pre- and post-conditions on recursive calls. This involves choosing a `PreRel` over recursive call events, `RPreInv`, and a `PostRel` over recursive call events, `RPostInv`. Just like any form of invariants in formal verification, correctly choosing `RPreInv` and `RPostInv` requires striking a careful balance between choosing preconditions that are weak enough to hold, but strong enough to imply post conditions. The rule is expressed in the following code.

```

Theorem padded_refines_mrec : forall (init1 : D1) (init2 : D2),
  RPreInv init1 init2 →
  (forall d1 d2, RPreInv d1 d2 →
    padded_refines (SumRel RPreInv RPre)
      (SumPostRel RPostInv RPost)
      (RPostInv d1 d2)
      (bodies1 d1) (bodies2 d2)) →
  padded_refines RPre RPost (RPostInv init1 init2)
    (mrec_spec bodies1 init1)
    (mrec_spec bodies2 init2).

```

The hypotheses in this theorem state that the initial recursive calls, `init1` and `init2`, are in the precondition `RPreInv`, and that given any two recursive calls related by the precondition, `d1` and `d2`, the recursive function `bodies` refine one another, where recursive calls are related by `RPreInv` and `RPostInv` and any other events are related by `RPre` and `RPost`. These reasoning principles allow us to prove complicated propositions involving the coinductively defined refinement relation without needing to perform direct coinduction.

While we include several parameter relations with the definition of `padded_refines`, at the top level, we are typically interested in the case where all relations are set to equality. We call this relation *strict refinement*, and refer to it with the \leq symbol.

30:18 Interaction Tree Specifications

```
Notation "phi1 '≤' phi2" :=
  (padded_refines eq PostRelEq eq phi1 phi2).
```

Strict refinement is a transitive relation, and is strong enough to allow rewrites under the context of any other application of `padded_refines`.

4.4 ITree specification Incompleteness

One way to interpret ITree specifications is as sets of ITrees. The following code defines *concrete* ITree specifications, which correspond to executable ITrees.

```
Variant concreteF {E R} `{EncodingType E} (F : itree_spec E R → Prop) : itree_spec E
  R → Prop :=
  | concreteRet (r : R) : concreteF F (Ret r)
  | concreteTau (t : itree_spec E R) : F t → concreteF F (Tau t)
  | concreteVis (e : E) (k : response_type e → itree_spec E R) :
    (forall a, F (k a)) → concreteF F (Vis (Spec_vis e) k).
Definition concrete {E R} `{EncodingType E} : itree_spec E R → Prop := gfp concreteF.
```

A concrete ITree specification contains no quantifiers along any of its branches. We can map each ITree specification to the set of *concrete* ITree specifications that refine it.

However, ITree specifications are not complete with respect to this interpretation. In particular, there are pairs of ITree specifications that represent equivalent sets of concrete ITree specifications, but do not refine one another. To see why, consider the following two ITree specification over an empty event signature `voidE`.

```
Definition top1 : itree_spec voidE unit :=
  forall_spec void;; Ret tt.
```

```
Definition top2 : itree_spec voidE unit :=
  or_spec spin (Ret tt).
```

Both `top1` and `top2` are refined by all concrete ITree specifications of type `itree_spec voidE unit`. We can prove the refinement for `top1` by applying the right `forall` rule, and reducing to a trivially satisfied proposition. For `top2`, we know that every concrete ITree specification of this type is `eutt` to either `spin` or `Ret tt`⁵. In each case, apply the right `exists` rule and choose the corresponding branch. However, given any relations `RE`, `REAns`, `RR`, we cannot prove `padded_refines RE REAns RR top1 top2`. This is because the only way to eliminate the `Spec_forall` on the left is to provide an element of the `void` type, which does not exist. This, along with the transitivity theorem, demonstrates that `padded_refines` is strictly weaker than the subset relation on sets of refining concrete ITree specification.

5 Total Correctness Specifications

This section discusses how to encode and prove simple pre- and post- condition specifications using ITree specifications. We also discuss how these definitions relate to our syntax-directed proof automation.

Suppose we have a program that takes in values of type `A` and returns values of type `B`. Suppose we want to prove that if given an input that satisfies a precondition `Pre : A → Prop`, it will return a value that satisfies a postcondition `Post : A → B → Prop` without triggering any other events. The postcondition is a relation over `A` and `B` to allow the postcondition to depend on the initial provided value. We can encode these conditions in the following ITree specification.

⁵ Proving this fact requires a nonconstructive axiom like the Law of The Excluded Middle.

```
Definition call_spec (a : A) : itree_spec (callE A B + E) B := trigger (inl (Call a)).
```

```
Definition calling' {F} {EncodingType F} : (A → itree F B) →
  (forall (c : callE A B) , itree F (response_type c)) :=
  fun f c => f (unCall c).
```

```
Definition rec_spec (body : A → itree_spec (callE A B + E) B) (a : A) :
  itree_spec E B :=
  mrec_spec (calling' body) (Call a).
```

```
Definition rec_fix_spec
  (body : (A → itree_spec (callE A B + E) B) → A →
   itree_spec (callE A B + E) B) :
  A → itree_spec E B :=
  rec_spec (body call_spec).
```

■ **Figure 11** `rec_fix_spec` Definition.

```
Definition total_spec : A → itree_spec E B :=
  fun a => assume_spec (Pre a);;
  b ← exists_spec B;;
  assert_spec (Post a b);;
  Ret b.
```

The specification assumes that the input satisfies the precondition, existentially introduces an output value, asserts the post condition holds, and finally returns the output.

The `total_spec` specification can be effectively used compositionally. Consider a merge sort implementation, named `sort`, built on top of two recursively defined helper functions, one for splitting a list in half, named `halve`, and one for merging sorted lists, named `merge`. If we have already proven specializations of `total_spec` for these sub functions, it becomes easier to prove a specification for `sort`. Immediately we can replace these sub functions with their total correctness specification. Now consider how this total correctness specification will behave on the left side of a refinement. First, we can eliminate `assume_spec (Pre a)` as long as we can prove `Pre a`. Once we have done that, we get to universally introduce the output `b`, along with a proof that it satisfies the post condition. We are finally left with only `Ret b` with the assumption `Post a b`. This is a much simpler specification than our initial executable specification, which relied on several control flow operators including a recursive one.

However, this easy to use specification is not easy to directly prove. The `padded_refines_mrec` rule gives us a sound reasoning principle for proving that a recursively defined function refines another recursively defined function, but it does not give any direct insight into how to prove any refinement that does not match that syntactic structure. To address this, we introduce a recursively defined version of `total_spec_fix` that we can apply our recursive reasoning principle on.

First, we introduce a specialization of the `mrec_spec` combinator called `rec_fix_spec`, defined in Figure 11. The `rec_fix_spec` function has a type similar to that of a standard fixpoint operator. The first argument, `body`, is a function that takes in a type of recursive calls `A → itree_spec (callE A B + E) B` and an initial argument of type `A` and produces a result in terms of an ITree specification. It relies on the `calling'` function to transform this value into a value of type `forall (c : callE A B) , itree_spec (callE A B + E) B` which the `mrec_spec` function requires. From there it relies on the `call_spec` and `rec_spec` functions to wrap values of type `A` into `Call` events and `trigger` them.

Given this recursion operator, we introduce an equivalent version of the total correctness specification, `total_spec_fix`.

30:20 Interaction Tree Specifications

```
Definition total_spec_fix : A → itree_spec E B :=
  rec_fix_spec (fun rec a ⇒
    assume_spec (Pre a);;
    n ← exists_spec nat;;
    trepeat n (
      a' ← exists_spec A;;
      assert_spec (Pre a' ∧ Rdec a' a);;
      rec a'
    );;
    b ← exists_spec B;;
    assert_spec (Post a b);;
    Ret b).
```

This specification is reliant on the `trepeat n t` function, which simply binds an ITree, `t`, onto the end of itself `n` times. Note that `total_spec_fix` is defined recursively, and contains the elements of `total_spec` inside the recursive body. This makes it easier to relate to recursively defined functions. It begins by assuming the precondition and ends by introducing an output, asserting it satisfies the post condition, and returning the output. What comes between these familiar parts requires more explanation. Recall the discussion of the `padded_refines_mrec` rule. This reasoning principle lets you prove refinement between two recursively defined ITree specifications when a single layer of unfolding of each specification match up one to one with recursive calls.

This means that to have a useful, general, and recursively defined version of total correctness specification we need to allow our recursive definition for total correctness specification to choose the number of recursive calls the function requires. For this reason, `total_spec_fix` existentially introduces a number `n` that specifies how many recursive calls are needed for one level of unfolding of the recursive function starting at `a`. The specification then includes `n` copies of a specification that existentially chooses a new argument `a'`, asserts a predicate holds on it, and then recursively calls the specification on this new argument. This asserted predicate contains two parts. First, we assert the precondition. A correct recursively defined function should not call itself on an invalid input if given a valid input. Second, we assert that `a'` is *less than* `a` according to the relation `Rdec`. In order for `total_spec_fix` to actually be equivalent to `total_spec`, we need to assume that `Rdec` is well-founded⁶. The fact that `Rdec` is well-founded ensures that this specification contains no infinite chains of recursive calls. This allows us to prove that `total_spec_fix` refines `total_spec` as long as `Rdec` is well-founded.

```
Theorem total_spec_fix_correct :
  well_founded Rdec → forall (a : A), total_spec_fix a ≤ total_spec a.
```

This theorem allows us to initially prove refinement specifications for recursive functions using the `padded_refines_mrec` rule with `total_spec_fix` and then replace it with the easier to work with `total_spec`.

Both `total_spec` and `total_spec_fix` do not accept any ITree specifications that trigger any events. As a result, these total correctness specifications do not allow any exceptions to be raised, as you would expect with total correctness specifications.

5.1 Demonstration

To demonstrate how to work with `total_spec`, we describe how to verify the `merge` function, a key component of the merge sort algorithm. The `merge` function takes two sorted lists and combines them into one larger sorted list which contains all the original elements. In

⁶ We use the Coq standard library's definition of well-foundedness for this.


```

Definition merge : (list nat * list nat)
  →
  itree_spec E (list nat) :=
  rec_fix_spec (fun rec '(l1,l2) ⇒
    b1 ← is_nil l1;;
    b2 ← is_nil l2;;
    if b1 : bool then
      Ret l2
    else if b2 : bool then
      Ret l1
    else
      x ← head l1;;
      tx ← tail l1;;
      y ← head l2;;
      ty ← tail l2;;
      if Nat.leb x y then
        l ← rec (tx, y::ty);;
        Ret (x :: l)
      else
        l ← rec (x::tx, ty);;
        Ret (y::l)).

Definition merge_pre p :=
  let '(l1,l2) := p in
  sorted l1 ∧ sorted l2.
Definition merge_post '(l1,l2) l :=
  sorted l ∧ Permutation l (l1 ++ l2).

Definition rdec_merge '(l1,l2) '(l3,l4) :=
  length l1 < length l3 ∧
  length l2 = length l4 ∨
  length l1 = length l3 ∧
  length l2 < length l4.

Theorem merge_correct : forall l1 l2,
  merge (l1,l2) ≤ total_spec merge_pre
  merge_post (l1,l2).

```

■ **Figure 12** Merge implementation.

Figure 12, we present a recursively defined implementation of `merge` along with relevant relations and the correctness theorem. The `merge` function is based on the standard list manipulating functions `is_nil`, `head`, and `tail`. We assume that the event type `E` contains some kind of error event which is emitted if `head` or `tail` is called on an empty list.⁷

The `merge` function relies on its arguments being sorted and guarantees that its output is a single, sorted list that is a permutation of the concatenation of the original lists. We formalize these conditions in `merge_pre` and `merge_post`. To prove that `merge` is correct, we want to show that it refines the total specification built from its pre- and post- conditions. To accomplish this, it suffices to choose a well founded relation and prove that `merge` satisfies the resulting `total_spec_fix` specification. For this function, we use `rdec_merge` which ensures that the pairs of lists that we recursively call `merge` on either both decrease in length, or one decreases in length and the other has the same length.

This leaves us with a refinement goal between two recursively defined specifications. We can then apply the `padded_refines_mrec_spec` theorem. For the relational precondition, we require that each pair of `Call` events is equal, and that `Pre` holds on the value contained within the call. For the relational postcondition, we require that equal `Call` events return equal values and that `Post` holds on them. Finally, we can prove that the body `merge` refines the body of `total_spec_fix` given these relation pre- and postconditions. We accomplish this by setting the existential variables on the right to make a single recursive call and give it the same argument as the recursive call that the body of `merge` makes.

With this technique, we can verify the simple server introduced in Section 1. Recall that the `server_impl` program executes an infinite loop of receiving a list of numbers, sorting it, and sending it back as a message. To verify `server_impl`, we first verify `halve`, the remaining sub function of `sort`, using the same technique we used to prove the correctness of `merge`. We can then use these facts to prove the correctness of `sort`, and use the correctness of `sort` to prove the correctness of `server_impl`.

```

Theorem server_correct :
  (server_impl tt) ≤ (server_spec tt).

```

⁷ We manage this assumption with a Coq type class called `ReSum`. For more information please read the original ITrees paper [29] or inspect the associated artifact.

30:22 Interaction Tree Specifications

Function Name	Description	C LoC	Proof LoC
<code>mbox_free_chain</code>	Deallocate an <code>mbox</code> chain	11	18
<code>mbox_len</code>	Compute the length in bytes of an <code>mbox</code> chain	9	40
<code>mbox_concat</code>	Concatenates an <code>mbox</code> chain after a single <code>mbox</code>	5	18
<code>mbox_concat_chains</code>	Concatenates two <code>mbox</code> chains	14	24
<code>mbox_split_at</code>	Split an <code>mbox</code> chain into two chains	25	147
<code>mbox_copy</code>	Copy a single <code>mbox</code>	13	74
<code>mbox_copy_chain</code>	Copy an <code>mbox</code> chain	18	173
<code>mbox_detach</code>	Detach the first <code>mbox</code> from a chain	18	18
<code>mbox_detach_from_end</code>	Detach the first N bytes from an <code>mbox</code> chain	3	50
<code>mbox_randomize</code>	Randomize the contents of an <code>mbox</code>	9	121
<code>mbox_drop</code>	Remove bytes from the start of an <code>mbox</code>	12	23

■ **Figure 13** Verified `mbox` functions.

6 Automation and Evaluation

6.1 Auto-active Verification

A key goal of this work is to provide auto-active automation for ITree specifications refinement. To this effect, the current section presents an automated Coq tactic for proving refinement goals called `prove_refinement`. The `prove_refinement` tactic is designed to reduce proof goals about refinement of programs to proof goals about the data and assertions used in those programs. In the spirit of auto-active verification, this is done mostly automatically, but with the user guiding the automation in places where human insight is needed.

The `prove_refinement` tactic defers to the user in two specific places. The first is in defining invariants for uses of the `mrec` recursive function combinator. The tool defers to the user to provide these invariants because inferring such invariants is undecidable. The second place where `prove_refinement` defers to the user is in proving non-refinement goals regarding first order data. The user can then apply other automated and/or manual proof techniques for the theories of the resulting proof goals.

The `prove_refinement` tactic is defined using a collection of syntax-directed inference rules for proving refinement goals. The tactic proves refinement goals by iteratively choosing and applying a rule that matches the current goal and then proceeding to prove the antecedents. The `prove_refinement` tactic implements this strategy using the Coq hint database mechanism, which is already a user-extensible mechanism for proof automation using syntax-directed rules.

We omit further implementation details both for space and because we do not claim the implementation of the `prove_refinement` tactic is novel or interesting. What is novel and interesting is that ITree specifications are designed in such a way that the straightforward implementation is able to achieve impressive results.

6.2 Evaluation

He et al. [9] discussed using Heapster to verify the interface of `mbox`, a key datastructure in the implementation of the Encapsulating Security Payload (ESP) protocol of IPSec. The `mbox` datastructure represents a data packet as a linked list of fixed length arrays. He et al. [9] type checked and extracted functional specifications for several functions that manipulate `mbox`. Using ITree specifications, we specified and verified the behavior of these functional

specifications using our auto-active verification tool. These functions are nontrivial, combining loops, recursion, and pointer manipulations. We present the list of verified functions in Figure 13.

For each function, we include the function’s name, a description of its behavior, the number of lines of C code in its definition, and the number of lines of Coq code required to verify it. Lines of code are, of course, a very coarse metric for judging the complexity of code and proofs. However, these metrics do demonstrate the viability of this verification approach, showing that the remaining proof burden after the automation is of a reasonable size. The primary advantage this approach has over others is that the system reduces the verification down to facts about first order data. In this case, the data is a variant of the `mbox` datastructure written in Coq.

7 Related Work

The most closely related work is the work on Dijkstra monads [16, 28, 1, 27]. Dijkstra monads are a framework for writing specifications over arbitrary monads. This framework is the basis for verifying programs with effects in F^* [26], a programming language specifically designed for verification. Dijkstra monads arise from the interaction of three structures, a *monad* M , a *specification monad* W , and an *effect observation* function `obs`. The monad M represents computations to be verified, while the specification monad W is a monad for writing specifications about those computations. The effect observation function `obs` is a monad homomorphism that embeds computations in M to the most precise specification in W that they satisfy. The specification monad is also equipped with a refinement relation that expresses when one specification implies or is contained in another. As an example, Dijkstra monads arose out of generalizing the notion of weakest precondition computations, by viewing the weakest precondition transformer of a computation as itself being a stateful computation from postconditions to preconditions. The mapping from a computation to its weakest precondition transformer is then a monad homomorphism from the computation monad to the weakest precondition monad.

ITree specifications in fact form a Dijkstra monad, where the type `itree_spec E R` acts as the specification monad and the corresponding ITree monad `itree E R` without logical quantifier events forms the computation monad. The effect observation homomorphism is then the natural embedding from the ITree type without quantifiers to the type with quantifiers. Most Dijkstra monads are specialized to act as either partial specification logics, which always accept any nonterminating computations, or total specification logics, which always reject any nonterminating computations. This means that most existing Dijkstra monads cannot reason about termination-sensitive properties like liveness. ITree specifications have the advantage of admitting specifications that accept particular divergent computations and not others. For example, an ITree specification could accept any computation that produces an infinite pattern of messages and responses from a server, and reject any computation that silently diverges.

A notable exception is the work of Silver and Zdancewic [25], who also provided a Dijkstra monad for ITrees. Much like ITree specifications it was capable of expressing specifications that allow for specifying infinite behavior. However, it did not provide reasoning principles for general recursion. The fact that ITree specifications represent specifications as syntax rather than semantics, as an ITree rather than some function relating ITrees to `Prop`, enabled us to write reasoning principles for general recursion and to build automation around the refinement rules.

A lot of work on verifying monadic computations has been based on notions of equational reasoning. This was in fact a key part of Moggi’s original work [19]. Pitts [21] and Moggi [20] extend this approach by building general theories of an evaluation predicate for reasoning about return values of computations. This approach provides no explicit means to reason about the effects, however, and also has no direct way of handling non-termination in specifications such as the specifications needed for a server process. Plotkin and Pretnar [22] further extend this approach with a general-purpose logic for algebraic effects, allowing it to reason about the effects themselves and not just return values. This approach cannot handle general Hoare logic assertions, however, and although there is a high-level discussion about handling recursion, it is not clear how well it works for those sorts of specifications. Rauch et al. [23] extends monads with native exceptions and non-termination and provides a logic for these monads. Much like in our work, monads in Rauch et al. [23] can be annotated with assertions. However, it restricts the language of assertions, and does not provide assumptions, or general universal or existential quantification. It also handles only tail recursive programs, and not general, mutual recursion.

One particularly effective approach in the space of equational reasoning was that of Gibbons and Hinze [8]. This work showed how to use the specialized monad laws of each sort of effect in a computation to define rewrite rules for simplifying and reasoning about effectful computations, and then demonstrated that this approach is both straightforward to use and powerful enough to verify a number of small but interesting programs.

The ultimate goal of this work is to provide techniques for auto-active verification of imperative code. Therefore, it is natural to compare this work to semi-automated separation logic tools like VST-Floyd[2] and CFML[6]. We argue this approach has two major advantages over these related techniques. First, while VST-Floyd is specialized to C and CFML is specialized to Caml, ITree specifications can be used to specify any programs with an ITrees based semantics. When paired with Heapster techniques, ITree specifications can be used to specify a wide array of imperative, heap-manipulating languages with a memory-safe type system. In particular, the Heapster type system is closely related to the Rust type system, meaning these techniques should be adaptable to specify and verify Rust code. Second, the Heapster types are able to perform all the separation logic specific reasoning, freeing the verifier to focus on the underlying mathematical structures.

8 Conclusion

This paper introduces ITree specifications along with verified metatheory and proof automation for reasoning about them. ITree specifications are a specialization of ITrees with a general notion of specification refinement. Unlike previous work developing specifications for ITrees, this paper provides techniques for working with the general recursion operator provided by the ITrees library. Finally, this paper demonstrates the effectiveness of its techniques by applying them on a collection of realistic C functions.

References

- 1 Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martinez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.
- 2 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, USA, 2014.

- 3 Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019.
- 4 Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.
- 5 Aaron R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.
- 6 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11*, pages 418–430, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2034773.2034828.
- 7 Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of amazon s2n. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, 2018.
- 8 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP)*, 2011.
- 9 Paul He, Edwin Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Stefanescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. A type system for extracting functional specifications from memory-safe imperative programs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2021.
- 10 Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022. doi:10.1145/3547647.
- 11 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013. doi:10.1145/2429069.2429093.
- 12 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM, 2016. doi:10.1145/2951913.2951943.
- 13 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09*, pages 207–220, New York, NY, USA, 2009. ACM. doi:10.1145/1629575.1629596.
- 14 Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, July 2008. doi:10.1007/s10817-008-9099-0.
- 15 Giuliano Losa and Mike Dodds. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, 2020.
- 16 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hrițcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341708.
- 17 Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. Rusthornbelt: A semantic foundation for functional verification of rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022.

- 18 Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. In *Proceedings of the 29th European Symposium on Programming (ESOP)*, 2020.
- 19 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS)*, 1989.
- 20 Eugenio Moggi. A semantics for evaluation logic. *Fundamenta Informaticae*, 22(1), 1989.
- 21 Andrew M. Pitts. Evaluation logic. In *Proceedings of the IV Higher Order Workshop*, 1990.
- 22 Gordon Plotkin and Matija Pretnar. A logic for algebraic effects. In *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2008.
- 23 Christoph Rauch, Sergey Goncharov, and Lutz Schröder. Generic hoare logic for order-enriched effects with exceptions. In Phillip James and Markus Roggenbach, editors, *Recent Trends in Algebraic Development Techniques*, pages 208–222, Cham, 2017. Springer International Publishing.
- 24 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15*, pages 77–87, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2737964.
- 25 Lucas Silver and Steve Zdancewic. Dijkstra monads forever: Termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434307.
- 26 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 266–278, 2011. doi:10.1145/2034773.2034811.
- 27 Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f^* . In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, pages 256–270, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837655.
- 28 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398, 2013. doi:10.1145/2491956.2491978.
- 29 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371119.

Breaking the Negative Cycle: Exploring the Design Space of Stratification for First-Class Datalog Constraints

Jonathan Lindegaard Starup  

Department of Computer Science, Aarhus University, Denmark

Magnus Madsen  

Department of Computer Science, Aarhus University, Denmark

Ondřej Lhoták 

David R. Cheriton School of Computer Science, University of Waterloo, Canada

Abstract

The λ_{DAT} calculus brings together the power of functional and declarative logic programming in one language. In λ_{DAT} , Datalog constraints are first-class values that can be constructed, passed around as arguments, returned, composed with other constraints, and solved.

A significant part of the expressive power of Datalog comes from the use of negation. Stratified negation is a particularly simple and practical form of negation accessible to ordinary programmers. Stratification requires that Datalog programs must not use recursion through negation.

For a Datalog program, this requirement is straightforward to check, but for a λ_{DAT} program, it is not so simple: A λ_{DAT} program constructs, composes, and solves Datalog programs at runtime. Hence stratification cannot readily be determined at compile-time.

In this paper, we explore the design space of stratification for λ_{DAT} . We investigate strategies to ensure, at compile-time, that programs constructed at runtime are guaranteed to be stratified, and we argue that previous design choices in the Flix programming language have been suboptimal.

2012 ACM Subject Classification Theory of computation \rightarrow Constraint and logic programming

Keywords and phrases Datalog, first-class Datalog constraints, negation, stratified negation, type system, row polymorphism, the Flix programming language

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.31

1 Introduction

Datalog is an expressive and powerful declarative logic programming language. A Datalog program consists of facts and rules. Facts represent knowledge (e.g. “an owl is a bird”) whereas rules allow one to infer new facts from existing facts (e.g. “if x is a bird and x is not a penguin then x can fly.”). The facts and rules imply a minimal model, a unique solution to every Datalog program [18] (e.g. “an owl is a bird” and “an owl can fly”).

Datalog has been used in a diverse set of applications including big-data analytics [20, 39, 41], social network analysis [39, 40], machine learning [32, 34], bio-informatics [25, 38], disassembly [15], micro-controller programming [46], networking and distributed systems [1, 10, 29], program analysis [7, 42, 43], and smart contract security [44].

Over the years, a plethora of Datalog extensions have been developed, adding support for object types [4], logic formulas [6], decidable arithmetic functions [23], disjunctive rule heads [13], distributed evaluation [24], and more. Many Datalog solvers have also been developed, including DLV [2], Soufflé [22], LogicBlox [3], QL [4], Formulog [6], and Flix [31].

A significant part of both the theoretical and practical expressive power of Datalog comes from the use of negation. However, negation also brings challenges: ensuring a meaningful semantics for logic programming with negation is a historically well-studied problem [14, 16, 17, 27, 36, 37]. Stratified negation has emerged as a simple and practical



© Jonathan Lindegaard Starup, Magnus Madsen, and Ondřej Lhoták;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 31; pp. 31:1–31:28

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

semantics that is accessible to ordinary programmers [48]. Informally, a Datalog program is stratified when there is no recursion through negation, i.e. no predicate symbol can negatively depend on itself. Stratification splits a Datalog program D into a sequence of Datalog programs D_1, \dots, D_n where the “output facts” of D_i become the “input facts” of D_{i+1} . Computing *whether* a Datalog program is stratified, and if so, computing its strata, is straightforward, e.g. using Ullman’s algorithm [45].

The λ_{DAT} calculus extends the lambda calculus with first-class Datalog constraints [30]. In λ_{DAT} , Datalog constraints, or programs, are values that can be constructed, passed as arguments to functions, returned from functions, composed with other Datalog program values, and have their minimal model computed. The minimal model is itself a set of facts, hence a Datalog value. This makes it possible to construct pipelines of Datalog computations. The type system of λ_{DAT} is based on Hindley-Milner [12, 47] extended with row polymorphism [28]. The type system permits Datalog constraints to be polymorphic, while ensuring type safety [30]. The λ_{DAT} calculus and its type system has been implemented in the Flix programming language.

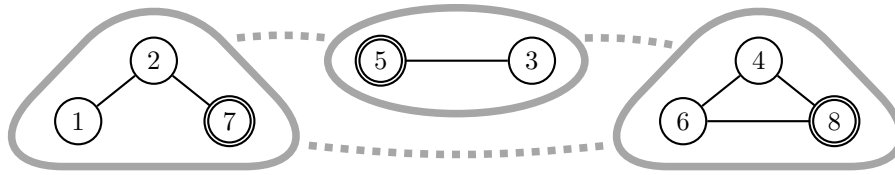
For a λ_{DAT} program, we cannot readily determine whether the Datalog values constructed at runtime are stratified. We *can* defer stratification until runtime, but this has two significant downsides: (1) we must perform the stratification repeatedly, and worse, (2) we have to abort execution if a non-stratified Datalog program is ever constructed.

In this paper, we explore the design space of compile-time techniques, which ensure that λ_{DAT} programs never construct non-stratified Datalog values at runtime. We also show that these techniques enable stratification of Datalog programs with lattice semantics.

In summary, the paper makes the following contributions:

- **(Design Space)** We explore the design space of compile-time stratification in the presence of first-class Datalog constraints.
- **(Framework)** We formulate the design space in a mathematical framework that allows us to express each design point as a specific instantiation of the framework. We introduce the notion of a *labelled dependency graph* and discuss how it can be used to soundly over-approximate the dependency edges of a Datalog expression.
- **(Comparison)** We identify the current state-of-the-art, i.e. the technique currently used in the Flix programming language, in the design space and illustrate that some of its design choices have been sub-optimal.
- **(Implementation)** We extend the Flix programming language with different design choices that admit more programs (i.e. allow more safe programs to pass the type checker). In particular, our extension uses rule-level granularity (**choice 1c**), and uses predicate arity and predicate term types in the labelled dependency graph (**choice 2b** and **2c**).
- **(Case Study)** We conduct a case study of a graph library that we implement in Flix. The study shows that use of stratified negation and lattice semantics is prevalent.

This paper is structured as follows: We motivate our work in Section 2. In Section 3 we present background material on Datalog, on stratified negation, and on the λ_{DAT} calculus. We explore the design space of stratification for λ_{DAT} in Section 4, Section 5, and Section 6. In Section 7 we discuss the design choices that we have made and our implementation in the Flix programming language. We use this implementation for a graph library case study in Section 8. Section 9 presents related work and Section 10 concludes. To ensure that the paper is self-contained, the background section has to cover a lot of material. Readers who are already familiar with Datalog, stratified negation, or the λ_{DAT} calculus are encouraged to skip the background material.



■ **Figure 1** An undirected graph. We want to compute the connected components of the graph and introduce edges that connect them. The components are indicated by thick gray lines. The edges we want to compute are indicated by dashed lines. The double circled nodes are the representatives of each connected component.

2 Motivation

We motivate our work with an example. Consider the following problem:

Given an undirected graph, compute its connected components and introduce an edge between each component.

Figure 1 illustrates the problem with an example. The graph has nodes numbered one to eight. The connected components are $\{1, 2, 7\}$, $\{3, 5\}$, and $\{4, 6, 8\}$. We want to compute the three (undirected) edges: $\{1, 2, 7\} \leftrightarrow \{3, 5\}$, $\{1, 2, 7\} \leftrightarrow \{4, 6, 8\}$, and $\{3, 5\} \leftrightarrow \{4, 6, 8\}$, as shown in Figure 1.

We can use the Flix programming language, with its support for first-class Datalog constraints, to elegantly solve this problem. Figure 2 shows a Flix program that does so. The program consists of two functions: `connectedComponents` and `connectGraph`.

The `connectedComponents` function computes the connected components (CCs) of the given (undirected) graph. The graph is represented as a set of `nodes` (of type `Set[Int32]`) and a set of `edges` (of type `Set[(Int32, Int32)]`). First, the function converts the `nodes` and `edges` into `Node` and `Edge` facts. Next, the function defines a local variable `r` which is a Datalog program value. The Datalog program defines `Reachable` as the (undirected) transitive closure of the `Edge` relation. Using `Reachable`, it computes the representative of each node in a CC as follows: Every node in a CC is associated with the lexicographically largest node in the same CC (`ReachUp`). The representative of a CC is then the node which is the largest in each CC, i.e. has no larger parent. Finally, the `connectedComponents` function composes the node (`ns`) and edge (`es`) facts with the Datalog program (`r`), computes its minimal model, and extracts all the `ComponentRep` facts. The polymorphic row extension in the return type `#{... | r}` allows the caller to type the returned Datalog program with additional predicates.

The `connectGraph` function computes a set of edges that connect CCs in a graph. That is, the function returns a set of edges that connect *sets of nodes*. The `connectGraph` function takes a graph as input (using the same representation as before), and calls the `connectedComponents` function to compute the CCs, specifically the `ComponentRep` relation which holds the representative of each node in the graph. The function defines the local variable `d` which is a Datalog program value. The Datalog program uses the `ComponentRep` to build a map `Component` which maps each representative to the set of nodes it represents. The rule:

```
1 Component(rep; Set#{n}) :- ComponentRep(n, rep).
```

states that if there is a `ComponentRep(n, rep)` fact, for some `n` and `rep`, then we infer a `Component(rep; Set#{n})` *lattice* fact where `rep` is mapped to the singleton set $\{n\}$. The `Component` lattice implicitly combines facts using the ordering on `Set[Int32]`, which is subset

31:4 Breaking the Negative Cycle

```
1  /// Given an undirected graph represented by nodes and edges,
2  /// computes its connected components and returns a relation
3  /// that maps each node to its representative.
4  def connectedComponents(nodes: Set[Int32], edges: Set[(Int32, Int32)]):
5      #{ ComponentRep(Int32, Int32) | r } =
6      let ns = inject nodes into Node;
7      let es = inject edges into Edge;
8      let r = #{
9          // Reachable(n1, n2) captures that n1 can reach n2.
10         // All nodes can reach themselves.
11         Reachable(n, n) :- Node(n).
12         // n1 can reach n2 directly.
13         Reachable(n1, n2) :- Edge(n1, n2).
14         // n2 can reach n1 directly, since the graph is undirected.
15         Reachable(n2, n1) :- Edge(n1, n2).
16         // n1 can reach n3 by transitivity.
17         Reachable(n1, n3) :- Reachable(n1, n2), Reachable(n2, n3).
18         // ReachUp contains nodes that can reach at least one other node
19         // with a higher value. That is, ReachUp contains all nodes which
20         // are not a representative of their connected component.
21         ReachUp(n1) :- Reachable(n1, n2), if n1 < n2.
22         // The representative, rep, of a node, n, in a connected component
23         // is any reachable node that is not contained in ReachUp.
24         ComponentRep(n, rep) :- Reachable(n, rep), not ReachUp(rep).
25     };
26     solve ns, es, r project ComponentRep
27
28  /// Given an undirected graph represented by nodes and edges,
29  /// connects all connected components. The returned edges are
30  /// between components, i.e. they are edges between *sets* of nodes.
31  def connectGraph(nodes: Set[Int32], edges: Set[(Int32, Int32)]):
32      #{ Edge(Set[Int32], Set[Int32]) | r } =
33      let d = #{
34          // Component(rep; c) captures that the node rep is the
35          // representative of the component c which is a set of nodes. The
36          // semicolon makes c use lattice semantics which aggregates all
37          // the nodes represented by rep into one set.
38          Component(rep; Set#{n}) :- ComponentRep(n, rep).
39          // Introduce an edge between every pair of components sets.
40          // The fix keyword ensures that the Component relation is fully
41          // materialized before this rule is evaluated, i.e. that a
42          // component contains all its nodes.
43          Edge(c1, c2) :- fix Component(_; c1), fix Component(_; c2).
44      };
45      solve connectedComponents(nodes, edges), d project Edge
46
47  def main(): Unit \ IO =
48      let connectedGraph = connectGraph(Set.range(1, 9),
49          Set#{ (1, 2), (2, 7), (5, 3), (8, 6), (6, 4), (4, 8)});
50      let result = query connectedGraph select (c1, c2) from Edge(c1, c2);
51      println(result)
```

■ **Figure 2** Flix program that connects an undirected graph using Datalog.

inclusion. In other words, if there are multiple `ComponentRep` facts with the same `rep` then every set is union'ed together. The last rule introduces edges between the components:

```
1  Edge(c1, c2) :- fix Component(_; c1), fix Component(_; c2).
```

The `fix` keyword ensures that the `Component` lattice *is fully computed* before the rule is evaluated. We will explain the full details in Section 6.

2.1 Stratified Negation

The Flix program in Figure 2 uses negation in the following rule:

```
ComponentRep(n, rep) :- Reachable(n, rep), not ReachUp(rep).
```

where the `ReachUp` predicate symbol occurs negated.

Datalog has the theoretically interesting and practically useful property that every Datalog program has a unique solution; the minimal model. However, in the presence of negation, an additional side condition is necessary to ensure the existence of the minimal model.

To understand why, consider a Datalog program with the two rules:

```
WinBlack(x)  $\Leftarrow$  not WinWhite(x). WinWhite(x)  $\Leftarrow$  not WinBlack(x).
```

The rules try to capture the idea that “if x is not a winning move for white then x is a winning move for black” and “if x is not a winning move for black then x is a winning move for white”. The problem is that this Datalog program has *two* models, neither of which is minimal: $\{\text{WinBlack}(p)\}$ and $\{\text{WinWhite}(p)\}$ for some p . We want to avoid such situations.

Stratified negation overcomes this problem by imposing a simple restriction: A predicate symbol cannot negatively depend on itself. This requirement is sometimes expressed as “no recursion through negation”. It is straightforward to determine if a Datalog program is stratified: We simply compute the dependency graph of the program and determine if it contains a cycle with a negative edge.

Returning to Figure 2, if we look at all the rules, we can see the following negative cycle:

```
Edge  $\Leftarrow$  Component  $\Leftarrow$  ComponentRep  $\Leftarrow$  ReachUp  $\Leftarrow$  Reachable  $\Leftarrow$  Edge
```

But does the Flix program in Figure 2 actually construct a Datalog value with a negative cycle at runtime? Fortunately this is not the case. The reason is as follows: the `Edge`, `Reachable`, and `ReachUp` predicates are used to compute the `ComponentRep` relation. This Datalog program is fully solved before `ComponentRep` is used to compute a new set of `Edges`.

We can now describe the problem this paper aims to solve:

In Flix, in the presence of first-class Datalog values, how can it be statically guaranteed that every Datalog value that may be constructed at runtime will be stratified?

In the example above, we used very ad-hoc reasoning to justify that a negative cycle cannot occur. While a very powerful and precise control- and data-flow analysis may be able to provide similar justification, in this paper we are interested in simpler and more light-weight techniques. We want to build on the type system of the λ_{DAT} calculus and of Flix. In the dependency `Edge \Leftarrow ComponentRep`, the type of `Edge` is $(\text{Set}[\text{Int32}], \text{Set}[\text{Int32}])$, whereas in the dependency `Reachable \Leftarrow Edge`, the type of `Edge` is $(\text{Int32}, \text{Int32})$. The type system ensures that predicate symbols with different types cannot occur in the same Datalog program value. This means that we can exclude the existence of edges based solely on the type information. Interestingly, we could also use the type system in a different way to exclude the negative cycle. The rule using negation mentions three predicate symbols: `ComponentRep`, `Reachable`, and `ReachUp`. The rule which closes the supposed cycle mentions two predicate symbols: `Component` and `Edge`. Since the program has no expression with a type that contains *all* these predicate symbols, we know that the cycle cannot occur.

We can now summarize the overall goal of this paper:

We want to explore the design space of type-based techniques that can statically ensure that Flix programs, in the presence of first-class Datalog constraints, are stratified.

3 Background

We begin with an introduction to Datalog and stratified negation before we move on to describe the λ_{DAT} calculus [30]. This paper contains a lot of background material. Readers who are already familiar with Datalog are encouraged to jump to Section 3.3.

3.1 Datalog

We give a brief introduction to Datalog. A comprehensive introduction is available in [9, 18].

3.1.1 Syntax

A Datalog *program* D is a set of constraints C_1, \dots, C_n . A *constraint* is of the form $A_0 \Leftarrow B_1, \dots, B_n$ where A_0 is the *head atom* and each B_i is a *body atom*. A head atom $p(t_1, \dots, t_n)$ consists of a predicate symbol p and a sequence of terms t_1, \dots, t_n . A *body atom* is similar to a head atom, except that it can be negated, which is written with “not” in front of the predicate symbol. A constraint without a body is called a *fact*. A constraint with a body is called a *rule*. A *term* is either a variable x or a literal constant c . An atom without variables is said to be *ground*. A fact or rule with only ground atoms is said to be ground. Figure 3 shows the grammar of Datalog.

$D \in \text{Programs}$	$= C_1, \dots, C_n$	$t \in \text{Terms}$	$= x \mid c$
$C \in \text{Constraints}$	$= A_0 \Leftarrow B_1, \dots, B_n.$	$c \in \text{Literals}$	$=$ a set of literal constants.
$A \in \text{Head Atoms}$	$= p(t_1, \dots, t_n)$	$x, y \in \text{VarSym}$	$=$ a set of variable symbols.
$B \in \text{Body Atoms}$	$= p(t_1, \dots, t_n)$	$p, q \in \text{PredSym}$	$=$ a set of predicate symbols.
	$ \text{ not } p(t_1, \dots, t_n)$		

■ **Figure 3** Syntax of Datalog.

A Datalog program must satisfy three syntactic properties that are not naturally captured by the grammar in Figure 3: (i) every fact must be ground, (ii) every variable that occurs in the head of a rule must also occur in its body, and (iii) every variable that occurs in a negated body atom must also occur in at least one positive body atom of the rule. If a program satisfies these properties it is said to be *well-formed*. In addition, every Datalog program which uses negation must be stratified, as we will explain in Section 3.2.

3.1.2 Semantics

The meaning of a Datalog program is usually defined in terms of the *minimal model*: the smallest interpretation (i.e. set of facts) that satisfy all the constraints (i.e. rule instantiations) of the program [18]. While the semantics of Datalog – and logic programs in general – is an interesting subject worthy of study, in this paper our focus is on stratification.

3.2 Stratified Negation

A significant part of the expressive power of Datalog comes from the use of negation, but unrestricted use of negation poses problems. Recall the Datalog program from Section 2:

$$\text{WinBlack}(x) \Leftarrow \text{not WinWhite}(x). \quad \text{WinWhite}(x) \Leftarrow \text{not WinBlack}(x).$$

If the program contains the constant 42, then the program has *two* solutions (models):

$$M_1 = \{\text{WinBlack}(42)\} \quad \text{and} \quad M_2 = \{\text{WinWhite}(42)\}$$

Neither model is a subset of the other. Hence neither model is minimal. This breaks one of the fundamental properties of Datalog: that every program has a unique solution. Defining a consistent semantics for logic programming languages with negation has long been studied and many proposals have been made [14, 16, 17, 27, 36, 37]. Stratified negation has emerged as a particularly simple semantics that can be mastered by ordinary programmers [48].

Informally, a Datalog program is said to be *stratified* if its predicate symbols can be partitioned into a sequence of *strata* such that a predicate symbol in a stratum only depends on predicate symbols in the same or lower strata. Intuitively, stratification splits a Datalog program D into a sequence of sub-programs D_1, \dots, D_n such that the output of D_i becomes the input of D_{i+1} .

We can determine if a Datalog program is stratified by computing its dependency graph:

► **Definition 1** (Dependency Graph). *The dependency graph (also called the precedence graph) of a Datalog program D is a directed graph of predicate symbols that contains:*

- a positive edge $a \leftarrow b$ if D contains a rule where a is the predicate symbol of the head atom and b is a predicate symbol of a positive body atom, and
- a negative edge $a \leftarrow\!-\! b$ if D contains a rule where a is the predicate symbol of the head and b is a predicate symbol of a negative body atom.

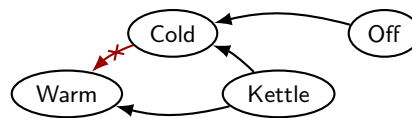
We write dependency edges as $a \leftarrow b$ and $a \leftarrow\!-\! b$ since this matches the “direction” of Datalog rules. We say that a depends on b . If $a \leftarrow\!-\! b$ we say that b must be computed before a . We write $\mathcal{DG}(D)$ for the dependency graph of the Datalog program D . Note that the dependency graph of a Datalog program D is unique.

We can now formally state when a Datalog program is stratified:

► **Definition 2** (Stratified). *A Datalog program D is stratified if its dependency graph contains no cycles with a negative edge.*

► **Example 3.** We will use the following running example. Consider the following Datalog program and its dependency graph:

$$\begin{aligned} \text{Cold}(x) &\leftarrow \text{Kettle}(x), \text{Off}(x). \\ \text{Warm}(x) &\leftarrow \text{Kettle}(x), \text{not Cold}(x). \end{aligned}$$



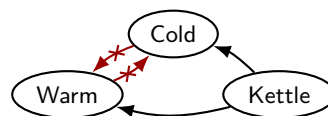
which does not contain a cycle with a negative edge. The strata of this program are:

$$s_1 = \{\text{Cold}, \text{Kettle}, \text{Off}\} \quad s_2 = \{\text{Warm}\}$$

which means that the Cold, Kettle, and Off relations must be computed before we compute the Warm relation.

► **Example 4.** Consider a modification of the previous Datalog program with its new dependency graph:

$$\begin{aligned} \text{Cold}(x) &\leftarrow \text{Kettle}(x), \text{not Warm}(x). \\ \text{Warm}(x) &\leftarrow \text{Kettle}(x), \text{not Cold}(x). \end{aligned}$$



31:8 Breaking the Negative Cycle

$ \begin{aligned} v \in \text{Val} &= c \mid \lambda x. e \mid \#\{C_1, \dots, C_n\} \\ e \in \text{Exp} &= x \mid v \mid e e \mid \text{let } x = e \text{ in } e \\ &\quad \mid e \langle + \rangle e \mid \text{solve } e \mid \text{project } p e \\ c \in \text{Literals} &= \text{a set of literal constants.} \\ x, y \in \text{VarSym} &= \text{a set of variable symbols.} \\ p, q \in \text{PredSym} &= \text{a set of predicate symbols.} \end{aligned} $	$ \begin{aligned} C \in \text{Constraints} &= A_0 \Leftarrow B_1, \dots, B_n. \\ A \in \text{Head Atoms} &= p(t_1, \dots, t_n) \\ B \in \text{Body Atoms} &= p(t_1, \dots, t_n) \\ &\quad \mid \text{not } p(t_1, \dots, t_n) \\ &\quad \mid \text{fix } p(t_1, \dots, t_n) \\ t \in \text{Terms} &= x \mid c \end{aligned} $
---	--

■ **Figure 4** Syntax of λ_{DAT} .

$ \begin{aligned} \tau \in \text{Type} &= \alpha \mid \iota \mid \tau_1 \rightarrow \tau_2 \mid r \\ r, s \in \text{Row} &= \rho \mid \{ \} \mid \{ p = (\tau_1, \dots, \tau_n) \mid r \} \\ \iota &= \text{a set of base types.} \end{aligned} $	$ \begin{aligned} \sigma \in \text{Scheme} &= \forall \bar{\alpha} \forall \bar{\rho}. \tau \\ \alpha \in \text{TypeVar} &= \text{a set of type variables.} \\ \rho \in \text{RowVar} &= \text{a set of row variables.} \end{aligned} $
---	---

■ **Figure 5** Type System of λ_{DAT} .

The graph contains a negative cycle between the **Cold** and **Warm** predicate symbols hence the program is not stratified and should be rejected. In this case, the negative cycle involves two predicate symbols and two negative dependencies, but in general a negative cycle consist of any number of dependency edges with at least one negative dependency edge.

3.3 First-Class Datalog Constraints

We now describe the λ_{DAT} calculus, a minimal lambda calculus with first-class Datalog constraints, originally introduced by [30]. We use a slightly simplified version of the calculus that illustrates the challenges posed by stratified negation.

3.3.1 Syntax

The grammar of λ_{DAT} is shown in Figure 4. The language includes the usual constructs from the lambda calculus: constants, variables, lambda abstractions, function applications, and let-bindings. Let-bindings support Hindley-Milner-style parametric polymorphism [11, 21, 33]. The values of λ_{DAT} include constants c , lambda abstractions $\lambda x. e$, and *Datalog values* $\#\{C_1, \dots, C_n\}$. A Datalog value is a collection of Datalog facts and rules. The grammar of Datalog values mirrors that of Figure 3. The fix body atom will be explained in Section 6. The expressions of λ_{DAT} include variables x , values v , function applications $e e$, and let-bindings $\text{let } x = e \text{ in } e$. The calculus has three expressions for working with Datalog values:

- (i) a *composition expression* $e_1 \langle + \rangle e_2$ to compute the union of two Datalog values,
- (ii) a *project expression* $\text{project } p e$ to extract all p facts from a Datalog value, and
- (iii) a *solve expression* $\text{solve } e$ to compute the minimal model of a Datalog value.

The Flix programming language, which implements the λ_{DAT} calculus, supports a richer set of operations for working with Datalog values. However, for our purposes, the above calculus is sufficient to illustrate the challenges. For the full details on the λ_{DAT} calculus, we refer the reader to [30].

3.3.2 Type System

The λ_{DAT} type system is based on Hindley-Milner [12, 47] extended with row polymorphism [28]. Each row type tracks the predicate symbols (and the types of the term parameters of each predicate) of a Datalog expression. The type system is sound; satisfying the usual progress and preservation theorems [30].

The type system splits types into mono- and poly types as shown in Figure 5. The mono types consist of type variables α , a set of base types denoted by ι (e.g. `Bool`), function types $\tau_1 \rightarrow \tau_2$, and row types r . A row type is either a row type variable ρ , an empty row $\{\}$, or a row extension $\{p = (\tau_1, \dots, \tau_n) \mid r\}$. A row type describes the type of a Datalog expression.

► **Example 5.** The following Datalog expression is typeable with the shown row type:

$$\#\{\text{Bird}(\text{“Eagle”}), \text{Flying}(x) \Leftarrow \text{Bird}(x), \text{not Penguin}(x).\} : \\ \{\text{Bird} = \text{String} \mid \{\text{Flying} = \text{String} \mid \{\text{Penguin} = \text{String} \mid \rho\}\}\}$$

The order of predicate symbols within a row is immaterial. Hence the same row is equivalent to (written as \cong):

$$\{\text{Penguin} = \text{String} \mid \{\text{Flying} = \text{String} \mid \{\text{Bird} = \text{String} \mid \rho\}\}\}$$

Figure 5 shows the poly types (or type schemes) of λ_{DAT} . A poly type is of the form $\forall \alpha_1, \dots, \alpha_n \forall \rho_1, \dots, \rho_m. \tau$. Thus, the λ_{DAT} calculus separates regular type variables α from row type variables ρ .

► **Example 6.** The following Datalog expression is typeable with the shown poly type:

$$\#\{\text{Path}(x, z) \Leftarrow \text{Path}(x, y), \text{Edge}(y, z).\} : \forall \alpha_1, \alpha_2 \forall \rho. \{\text{Path} = (\alpha_1, \alpha_2), \text{Edge} = (\alpha_2, \alpha_2) \mid \rho\}$$

This expression is polymorphic in the types of the terms of the `Edge` and `Path` atoms (α_1 and α_2) and row polymorphic in the type of the rest of the row (ρ). As can be seen from the rule, the variables y and z must share the same type (α_2) because of their occurrences in the `Path` atoms. The two types of polymorphism serve two different purposes: the regular polymorphism allows the expression to be used with terms of different types (e.g. `Edge` and `Path` facts over integers, strings, etc) whereas the row polymorphism allows the expression to be composed with other Datalog expressions that may have additional predicate symbols.

The type system of λ_{DAT} has three mutually inductive typing judgments: one for expressions ($\Gamma \vdash e : \tau$), one for constraints ($\Gamma \vdash_c C : r$), and one for atoms ($\Gamma \vdash_p p(t_1, \dots, t_n) : r$).

3.3.2.1 Type Rules

We briefly describe the (T-HEAD-ATOM) and (T-CONSTRAINT) type rules of λ_{DAT} .

The typing judgement $\Gamma \vdash_p p(t_1, \dots, t_n) : r$ captures that the head or body atom $p(t_1, \dots, t_n)$ can be typed with row type r under the type environment Γ . In particular, the

$$\frac{\forall i. \Gamma \vdash t_i^h : \tau_i}{\Gamma \vdash_p p(t_1^h, \dots, t_n^h) : \{p = (\tau_1, \dots, \tau_n) \mid r\}} \quad (\text{T-HEAD-ATOM})$$

rule states that a head atom can be typed as a row type in which the predicate symbol p is mapped to a tuple type whose elements are the types of the head terms t_1^h, \dots, t_n^h . The type rules for body atoms are similar. What is important, for our purposes, is that to type a head or body atom, its predicate symbol and term types must be part of the row type.

The typing judgement $\Gamma \vdash_c C : r$ captures that the constraint C can be typed as r under the type environment Γ . In particular, the

$$\frac{\Gamma, x_1 : \tau_1, \dots, x_m : \tau_m \vdash_p A_0 : r \quad \forall i. \Gamma, x_1 : \tau_1, \dots, x_m : \tau_m \vdash_p B_i : r_i \quad \forall i. r \cong r_i}{\Gamma \vdash_c \forall (x_1 : \tau_1, \dots, x_m : \tau_m). A_0 \Leftarrow B_1, \dots, B_n. : r} \quad (\text{T-CONSTRAINT})$$

31:10 Breaking the Negative Cycle

type rule states that to type an entire constraint, the row type of the head atom and all the body atoms must be equivalent, i.e. contain the same predicate symbols mapped to the same term types, modulo the order of predicate symbols. In λ_{DAT} , the unbound Datalog variables are explicitly quantified.

We refer the reader to [30] for a complete description of the type system. We will use the type system when we define soundness of labelled dependency graphs in Section 5.

3.4 The Problem: Stratification and First-Class Constraints

We are now ready to define what it means for a λ_{DAT} program to be stratified:

► **Definition 7** (Stratification for λ_{DAT}). *A λ_{DAT} calculus program P is stratified if every Datalog value constructed during evaluation of P is stratified.*

The challenge is to statically determine when that is the case. Consider the λ_{DAT} program:

$$f = \lambda c_1. \lambda c_2. \text{let } r = \#\{P(x) \Leftarrow A(x), \text{not } Q(x).\} \text{ in } c_1 \langle + \rangle c_2 \langle + \rangle r$$

To determine whether f returns a stratified program we must know at least:

- whether the argument c_1 is itself stratified,
- whether the argument c_2 is itself stratified,
- whether the composition of c_1 and c_2 is stratified, and finally,
- whether the composition of c_1 , c_2 with the rule r is stratified.

At run time, the values of c_1 , c_2 , and r are known and we can use Ullman’s algorithm [45] to compute their stratification. But again, moving the stratification check to run time would force the program to crash if it ever encounters a Datalog value that cannot be stratified!

Before we explore the design space of techniques to ensure compile-time stratification of λ_{DAT} calculus programs, let us pause and reflect on what makes a design “good”. As discussed, we are interested in techniques that are fully automatic, hence imposes no additional burden on the programmer. In addition, we want a system that: (i) has high precision (i.e. few programs are unfairly rejected), (ii) is fast (i.e. can be run continuously during program development), (iii) offers understandable error messages (i.e. does not require too much knowledge from the programmer “when things go wrong”), and (iv) is robust under refactoring (i.e. harmless refactorings should not break stratification). As is often the case, some of these goals are conflicting.

4 Dependency Graph Types: A Purely Type-based Approach

We now present a type system that captures the entire dependency graph in the type system itself. This type system is expressive, precise, and its types can be fully inferred. As we shall discuss, it is also impractical, since each type may be quadratic in the number of predicates.

We extend the λ_{DAT} type system to track the *entire* dependence graph of every Datalog expression in the type system. The key idea is straightforward: We represent a dependency edge $p \leftarrow q$ or $p \leftarrow\!-\! q$ as a single “label” and then use row polymorphism to track a row of all these labels. The type system does not concern itself with the types of terms and should be understood as being in addition to the existing type system.

The new row types are given by the grammar:

$$r = \rho \mid \{ \} \mid \{ p \leftarrow q \mid r \} \mid \{ p \leftarrow\!-\! q \mid r \}$$

The type rule for a Datalog constraint is straightforward:

$$\frac{A_0 = p_h(t, \dots, t) \quad E = \{p_h \leftarrow p_b^i \mid B_i = p_b^i(t, \dots, t)\} \cup \{p_h \leftarrow\! \times p_b^i \mid B_i = \text{not } p_b^i(t, \dots, t)\}}{\Gamma \vdash_e \forall(x_1 : \tau_1, \dots, x_m : \tau_m). A_0 \Leftarrow B_1, \dots, B_n. : \{E \parallel r\}}$$

where $\{E \parallel r\}$ is the row type with all be labels in the set E , e.g. $\{\{x, y\} \parallel r\} = \{x \mid \{y \mid r\}\}$. Intuitively, the type rule states that if we have a constraint $A_0 \Leftarrow B_1, \dots, B_n$ where the head predicate symbol is p_h and there is a positive body atom $B_i = p_b^i(t, \dots, t)$, then the row contains the positive edge $p_h \leftarrow p_b^i$. Similarly, if there is a negative body atom $B_i = \text{not } p_b^i(t, \dots, t)$, then the row contains the negative edge $p_h \leftarrow\! \times p_b^i$.

For this type system, we conjecture the important property:

► **Theorem 8 (Soundness).** *Let $\Gamma \vdash e : r$ and define e' to be e where all the free variables have been substituted for values of the types given in Γ . If $e' \rightarrow^* v$ then the dependency graph of v is a subset of g , i.e. $\mathcal{DG}(v) \subseteq g$, where g is the graph defined by the edges present in the row type of v .*

We now give two examples of how the type system works. We will use the abbreviations Warm (W), Kettle (K), Cold (C), and Off (O) moving forward:

► **Example 9.** The left expression is typeable with the abbreviated type on the right:

<pre>let p = #{ Cold(x) :- Kettle(x), Off(x). Warm(x) :- Kettle(x), not Cold(x). }</pre>	$\forall \rho. \{C \leftarrow K, C \leftarrow O,$ $W \leftarrow K, W \leftarrow\! \times C, \rho\}$
--	--

► **Example 10.** Consider the expressions on the left and their abbreviated types on the right:

<pre>let p1 = #{ Cold(x) :- Kettle(x), not Warm(x). }; let p2 = #{ Warm(x) :- Kettle(x), not Cold(x). };</pre>	$p_1 : \forall \rho_1. \{C \leftarrow K, C \leftarrow\! \times W \mid \rho_1\}$ $p_2 : \forall \rho_2. \{W \leftarrow K, W \leftarrow\! \times C, \rho_2\}$
--	--

The composition of p_1 and p_2 has the following row type which contains a negative cycle between W and C and is rejected.

$$\forall \rho_3. \{C \leftarrow K, C \leftarrow\! \times W, W \leftarrow K, W \leftarrow\! \times C, | \rho_3\}$$

4.1 Discussion

The type system has several advantages:

- it captures the dependency graph of each Datalog expression in its type,
- it supports row polymorphism, and
- it has complete type inference.

The type system is a simple and straightforward application of row types. But the strength of the type system is also its weakness: The type of each expression needs to store the whole dependency graph between all pairs of predicate symbols in the expression. The amount of information to be stored in each type is quadratic in the number of predicate symbols.

To understand why a type needs to store, for each pair of predicate symbols A and B , whether or not A is reachable from B in the dependency graph, consider the Datalog value $\#\{A(x) \Leftarrow \text{not } B(x)\}$. If this value is composed with another Datalog value v , the resulting Datalog program is stratified if and only if B does not depend on A in v . Since A and B could be arbitrary predicate symbols, the type of v needs to store, for every possible pair

31:12 Breaking the Negative Cycle

of predicate symbols (A, B) , whether or not there is a dependence in v . While we have not implemented the system, this complexity leads to concerns about efficiency of type inference. In particular, the rows used to track all dependency edges are now very long. Instead, we want to explore the design space of a hybrid approach: We keep the original type system of [30] and we combine it with global information about the constraints in the entire program.

5 Labelled Dependency Graph: A Hybrid Approach

We now describe a hybrid approach that combines local information from the type system with global information about the λ_{DAT} program. As it turns out, the choice of what information to collect about the entire program opens up a large design space.

General Framework

We want to statically ensure that a λ_{DAT} program is stratified in the sense of Definition 7. To do so, we take the following overall approach:

For each Datalog expression e in a well-typed λ_{DAT} program P (i.e. we have $\Gamma \vdash e : r$), we want to construct a dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ that soundly over-approximates the dependency graph(s) of every Datalog value v that e could evaluate to at runtime. In other words, define e' to be e where all the free variables have been substituted for values of the types given in Γ . These values must be chosen from compositions of the Datalog literals in the program. If $e' \rightarrow^* v$ then $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ over-approximates $\mathcal{DG}(v)$. The role of the parameter $\mathcal{LG}(P)$ will be discussed shortly.

If the dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ for the expression e is stratified, then every Datalog value v that the expression e could evaluate to must also be stratified: if the over-approximate dependency graph does not contain a negative cycle then any sub-graph cannot contain a negative cycle. If this is true for every expression e in a program P , then the entire program must be stratified in the sense of Definition 7.

We construct the over-approximate graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ using two types of information:

- Local information about the expression e (the $e : r$ part).
- Global information about the entire program P (the $\mathcal{LG}(P)$ part).

We call the data structure that records the global information the *labelled dependency graph* $\mathcal{LG}(P)$. The graph records all (positive and negative) dependencies between predicate symbols in all Datalog rules appearing anywhere in λ_{DAT} the program. The dependency edges are annotated with labels that record various constraints about each dependency. When constructing a specific dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ for a specific expression e of type r , local information about the expression recorded in the type r will be combined with the constraints recorded in the edge labels to determine that certain edges represent global dependencies that are incompatible with some characteristics of the local expression e . These edges from the global dependency graph are removed to construct a more precise local dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ specific to the local expression e .

Formally, we define the labelled dependency graph as:

► **Definition 11** (Labelled Dependency Graph). *The labelled dependency graph $\mathcal{LG}(P)$ of a λ_{DAT} program P is a directed graph between predicate symbols where each edge is labelled with information that is used to determine if that edge is possible w.r.t. to a specific row type.*

and we require that the \mathcal{LG} and $\widehat{\mathcal{DG}}$ functions satisfy the following important property:

► **Definition 12** (Soundness Criterion). *Given a well-typed λ_{DAT} program P , assume that e is a sub-expression of P and that $\Gamma \vdash e : r$. Define e' to be e where all the free variables have been substituted for values of the types given in Γ . These values must be chosen from compositions of the Datalog literals in P . If $e' \rightarrow^* v$ then the dependency graph $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ is a sound over-approximation of the dependency graph of v , i.e. $\mathcal{DG}(v) \subseteq \widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$.*

The choice of what information to record in the labelled dependency graph opens a large design space. For example, it could include information about the constraints that occur in the program, their predicate symbols, and the types of their terms. The design space contains various choices of possible constraints that can be recorded in the labels of the global dependency graph.

5.1 Design Choice 1: Granularity of the Labelled Dependence Graph

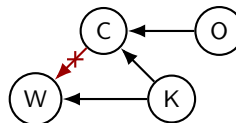
We now turn to the choice of which labels to use on the labelled dependency graph $\mathcal{LG}(P)$.

5.1.1 Degenerate

The simplest choice is to leave the labelled dependency graph unlabelled. This is a degenerate choice which corresponds to the most pessimistic assumption: that all Datalog values could be composed into one big Datalog value. We include it for completeness.

► **Example 13.** Consider the Datalog expression on the left:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



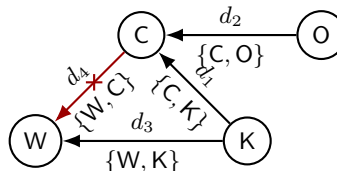
which gives rise to the labelled dependency graph on the right. The dependency edges carry no additional information and hence any local information about the type of a specific Datalog expression cannot help narrow down the set of possible edges. We have to consider all edges as possible.

5.1.2 Source and Destination Granularity

A straightforward improvement is to label each dependency edge with its source and destination predicate. This is information that is already represented by the graph itself.

► **Example 14.** Consider again the Datalog expression:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



which now has the labelled dependency graph on the right. Each edge is now labelled with its source and destination predicate symbols. We can use this information as follows.

Suppose we are given an expression e with type r :

$$r = \{ \dots \mid \text{Cold} = \dots \mid \text{Off} = \dots \mid \text{Warm} = \dots \mid \dots \}$$

where r does not contain the **Kettle** predicate symbol. If so, we know that e cannot evaluate to a Datalog value v which would give rise to the dependency edges d_1 and d_3 (because these

31:14 Breaking the Negative Cycle

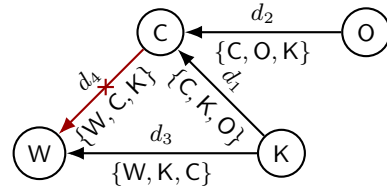
edges are labelled with the Kettle predicate symbols and the type system guarantees that e cannot evaluate to a Datalog value with a Kettle predicate). On the other hand, we cannot exclude the d_2 and d_4 edges because their labels occur in the type.

5.1.3 Rule-level Granularity

A more interesting design choice is to label each dependency edge with *all* predicate symbols that occur in the rule from which the edge originates.

► **Example 15.** Consider again the Datalog expression:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



which now has the labelled dependency graph on the right. Each dependency edge is now labelled with *all the predicate symbols that occur in the rule* from where the edge originates.

For example, the edge $C \xleftarrow{\{C, K, O\}} K$ represents that Cold depends on Kettle but it can only occur if Cold, Kettle, *and* Off can occur in the Datalog value.

Suppose, as before, that we are given an expression e with type r :

$$r = \{\dots \mid \text{Cold} = \dots \mid \text{Off} = \dots \mid \text{Warm} = \dots \mid \dots\}$$

where r does not contain the Kettle predicate. We are now able to exclude *all* dependency edges because they are all labelled with Kettle. Intuitively, the Datalog expression on the left uses the Kettle predicate in both rules. Thus, if a Datalog expression does not contain the Kettle predicate then neither rule can contribute to its dependency graph.

Suppose, on the other hand, that we are given an expression e_2 with type r_2 :

$$r_2 = \{\dots \mid \text{Cold} = \dots \mid \text{Kettle} = \dots \mid \text{Warm} \mid \dots\}$$

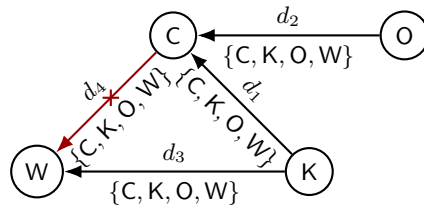
where r_2 does not contain the Off predicate. If e_2 evaluates to a Datalog value v_2 , we can exclude the dependency edges d_1 and d_2 from its dependency graph, but we cannot exclude d_3 nor d_4 . This is because we are able to exclude the dependency edges from the first rule, but not from the second.

5.1.4 Datalog Value-level Granularity

The last and most powerful option is to label each dependency edge with *all* predicate symbols that occur within the same Datalog literal.

► **Example 16.** Consider one last time the Datalog expression:

```
let p = #{
  Cold(x) :- Kettle(x), Off(x).
  Warm(x) :- Kettle(x), not Cold(x).
}
```



which now has the labelled dependency graph on the right. Each dependency edge is labelled with *all the predicate symbols that occur in the same Datalog literal*. For example, the edge $C \xleftarrow{\{C, K, O, W\}} K$ represents that Cold depends on Kettle but only if *all* the predicates Warm, Cold, Kettle, *and* Off can occur in the Datalog value.

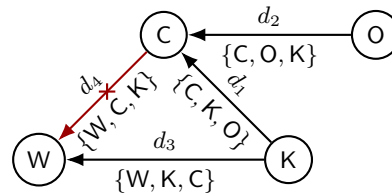
Suppose we are given an expression e with type r :

$$r = \{\dots \mid \text{Cold} = \dots \mid \text{Kettle} = \dots \mid \text{Warm} = \dots \mid \dots\}$$

which does not contain the `Off` predicate symbol. If so, we know that e cannot evaluate to a Datalog value v with *any* of the four dependency edges. Note, in particular, that we are able to exclude the dependency edge $W \leftarrow C$ even though the predicate symbol `Off` has nothing to do with `Warm` or `Cold` or even the rule from which the edge arises.

While this design choice is very powerful, it suffers from the problem that a simple refactoring can break stratification. For example, if we take the same program and refactor it to:

```
let p1 = #{
  Cold(x) :- Kettle(x), Off(x).
};
let p2 = #{
  Warm(x) :- Kettle(x), not Cold(x).
};
let pr = p1 <+> p2
```



then it is no longer the case that the rule in p_1 must occur together with p_2 . Consequently, if we have an expression e with the type r (as above), we can no longer exclude the dependency edges d_3 and d_4 . Thus, while this design choice is powerful, it is also brittle under refactoring.

5.1.5 Summary

In summary, the four design choices are:

- **Choice 1a:** The degenerate case where $\mathcal{LG}(P)$ is unlabelled.
- **Choice 1b:** Label the $\mathcal{LG}(P)$ with the source and destination predicate symbols from which the dependency edge arises.
- **Choice 1c:** Label the $\mathcal{LG}(P)$ with *all the predicate symbols that occur in the same rule* from where the dependency edge originates.
- **Choice 1d:** Label the $\mathcal{LG}(P)$ with *all the predicate symbols that occur in the same Datalog literal* from where the dependency edge originates.

For the purpose of exposition, we assume **choice 1c** for the next subsections.

5.2 Design Choice 2: Enriched Labelling and Type Filtering

We can increase precision by including information about the types of the predicate symbols in the labelled dependency graph.

5.2.1 Predicate Symbol Arity

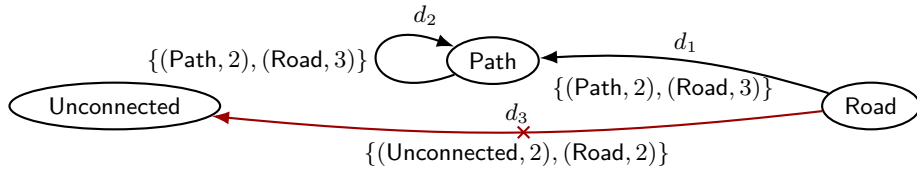
We can include the arity of the predicate symbols in the labelled dependency graph to add precision. We define the labels to be a set of pairs (p, n) of predicate symbols and their arity. We then define $\widehat{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ to include an edge $a \xrightarrow{\ell} b$ when the arities in ℓ agree with the arities in the row type r .

► **Example 17.** Consider a λ_{DAT} program P that contain the two Datalog expressions:

```
#{Path(x, y) :- Road(x, l, y).
  Path(x, z) :- Path(x, y), Road(y, l, z).}
#{Unconnected(x, y) :-
  ..., not Road(x, y).}
```

In the Datalog expression on the left, the `Road` predicate symbol has three terms whereas in the Datalog expression on the right, the `Road` predicate has two terms. The labelled dependency graph, $\mathcal{LG}(P)$, is:

31:16 Breaking the Negative Cycle



which includes the arity of each predicate in the labels on the edges. Suppose the program contains an expression e with the row type r :

$$r = \{\text{Road} = (\text{Int32}, \text{Int32}) \mid \text{Unconnected} = (\text{Int32}, \text{Int32}) \mid \dots\}$$

If the e evaluates to a Datalog value v , then the type system guarantees that every atom in v with the predicate symbol `Road` must have two terms, both of which are of type `Int32`. Thus we can exclude the two dependency edges d_1 and d_2 because the arities on the labels do not match.

5.2.2 Predicate Term Types

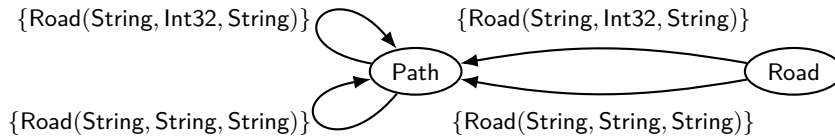
We can increase precision even further by including the term types of predicate symbols in the labelled dependency graph. We define labels to be pairs $(p, \bar{\tau})$ of a predicate symbol and the types of its terms, also written $p(\bar{\tau})$ on the labelled graphs. We then define $\overline{\mathcal{DG}}(e : r, \mathcal{LG}(P))$ to include an edge $a \xrightarrow{\ell} b$ when the term types in ℓ unify with term types in the row r .

► **Example 18.** Consider a λ_{DAT} program P that contain the two Datalog expressions:

<pre>#{Road("Lyon", 120, "Paris"). Path(x, z) :- Path(x, y), Road(y, l, z).}</pre>	<pre>#{Road("Lyon", "Icy", "Paris"). Path(x, z) :- Path(x, y), Road(y, l, z).}</pre>
--	--

In each Datalog expression, the `Road` predicate has arity two. In the expression on the left, the label on a `Road` fact represents the current weather (type `String`), whereas in the expression on the right, the label represents the current speed limit (type `Int32`).

The labelled dependency graph, omitting the `Path` predicate in labels, is:



5.2.3 Relational and Lattice Predicate Symbols

In Flix, as we shall discuss further in Section 6, every predicate symbol is given either a relational or a lattice interpretation. The type system ensures that relational and lattice predicate symbols cannot be mixed. That is, within a Datalog value every predicate symbol has exactly one interpretation. Similarly to how we extended the labelled dependency graph with arity and term types, we can also extend it to account for this information.

In summary, the four options are:

- **Option 2a:** Enrich $\mathcal{LG}(P)$ to track the arity of the predicates.
- **Option 2b:** Enrich $\mathcal{LG}(P)$ to track term types of the predicates.
- **Option 2c:** Enrich $\mathcal{LG}(P)$ to distinguish between relations and lattices.
- **Option 2d:** Do not enrich $\mathcal{LG}(P)$.

These options are not mutually exclusive and can be combined.

5.3 Choice 3: Stratify With or Without Monomorphization

We have shown how local information, i.e. information about the row type of an expression, enables us to filter the labelled dependency graph. An orthogonal design choice, which affects precision, is whether to perform stratification with or without monomorphization.

Monomorphization is a compile-time transformation that replaces polymorphic functions by copies that are specialized to their concrete type arguments. For example, if `List.map` is used with both integer and string lists, then monomorphization generates two copies of `List.map`: one specialized to integers and one specialized to strings. While the primary purpose of monomorphization is to avoid boxing, monomorphization can also be used to improve the precision of stratification.

Monomorphization improves precision in two ways:

- (i) by specializing polymorphic Datalog expressions to concrete types, boosting the precision of type-based filtering, and
- (ii) by eliminating unreachable Datalog expressions.

► **Example 19.** Consider the Flix program fragment:

```
def f(): #{A(t), B(t)} =
  #{ A(x) :- B(x). }

def g(): #{A(t), B(t)} =
  # { B(x) :- B(x), not A(x). }

def main(): Unit =
  let c1 = f() <+> A(123).;
  let c2 = g() <+> A("hello").;
  solve c1
```

The functions f and g return Datalog values with row types that contain the predicate symbols A and B , which are polymorphic in the type parameter t . The two Datalog constraints form a negative cycle between A and B . Because the types of the two expressions are

$$\forall\alpha_1, \forall\rho_1. \{A = \alpha_1 \mid \{B = \alpha_1 \mid \rho_1\}\} \quad \text{and} \quad \forall\alpha_2, \forall\rho_2. \{A = \alpha_2 \mid \{B = \alpha_2 \mid \rho_2\}\}$$

we cannot exclude that these two types could occur in the same Datalog value and consequently we cannot exclude that the overall program might construct a non-stratified Datalog program at runtime. However, if we monomorphize the program first, i.e. specialize f and g to their concrete type arguments whenever they are used in the program, then we obtain the program:

```
def f1(): #{A(Int32), B(Int32)} =
  #{ A(x) :- B(x). }

def g1(): #{A(String), B(String)} =
  # { A(x) :- A(x), not B(x). }

def main(): #{A(Int32), B(Int32)} =
  let c1 = f1() <+> A(123).;
  let c2 = g1() <+> A("hello").;
  solve c1
```

where $f1$ and $g1$ are no longer polymorphic and the types of their Datalog expressions are:

$$\{A = \text{Int32} \mid B = \text{Int32}\} \quad \text{and} \quad \{A = \text{String} \mid B = \text{String}\}$$

We can now use the types to determine that the two rules cannot occur in the same Datalog value and consequently the program is stratified.

Monomorphization increases precision, but has two practical downsides: the labelled dependency graphs are larger and consequently more costly to stratify, and intertwining monomorphization and stratification may make it difficult for programmers to understand why or when a program fails to be stratified.

In summary, the two design choices are:

- **Choice 3a:** Stratify *without* monomorphization.
- **Choice 3b:** Stratify *with* monomorphization.

6 The Fix Modifier, Lattice Semantics, and Stratification

We now explain the role of the fix modifier and its semantics. Intuitively, the fix modifier enables us to safely use lattice values in relations. Given the rule:

$$A(x) \Leftarrow \text{fix } B(x), C(x).$$

The use of “fix” in front of the atom $B(x)$ forces the relation B to be fully computed before the rule is applied. Therefore, A must belong to a stratum strictly greater than B . Thus, the fix modifier has the same effect on stratification as negation. For a normal Datalog program, fix does not change the minimal model, only the evaluation order. However, for Datalog programs with lattice semantics [31], the fix construct solves a long-standing problem.

In Flix, every predicate symbol is associated with a *relational* or *lattice* interpretation. We will write p for a predicate symbol that has a relational interpretation and p_ℓ for a predicate symbol that has a lattice interpretation. We syntactically distinguish between relational and lattice predicates by writing a relational predicate as $A(t_1, \dots, t_n)$, whereas we write a lattice predicate as $A(t_1, \dots, t_n; t)$, with a semi-colon before the last term.

► **Definition 20** (Key and Lattice Positions). *Given an atom $p(t_1, \dots, t_n)$ where p has relational interpretation, we say that the terms t_1, \dots, t_n are in key position. Given atom $p_\ell(t_1, \dots, t_n; t)$ where p_ℓ has a lattice interpretation, we say that the terms t_1, \dots, t_n are in key position and t is in lattice position.*

► **Definition 21** (Key and Lattice Variables). *We split variables into key and lattice variables. A variable is a key variable if all its occurrences are in key positions. Otherwise it is a lattice variable.*

As the definition states, a variable that occurs in *both* a key and lattice position is considered a lattice variable. The original version of Flix disallows such “dual-use” of variables; enforcing that lattice variables cannot be used in key position.

► **Example 22.** To better understand key and lattice variables, consider the Datalog rules:

```
A(k1, k2; l) :- B(k1, k2; l), C(k1, k2; l). // legal
A(k1, k2, l) :- B(k1, k2; l), C(k1, k2; l). // illegal
A(k1, k2; l) :- B(k1, k2; l), C(k1, k2, l). // illegal
A(k1, l; k2) :- B(k1, k2; l), C(k1, k2; l). // illegal
```

In each rule, the variable l is a lattice variable because it occurs in at least one lattice position. The first rule is legal since the lattice variable l only occurs lattice positions. The second rule is illegal since the lattice variable l occurs in a key position in the head of the rule (where A has a relational interpretation). The third rule is illegal since the lattice variable l occurs in a key position in the body of the rule (where C has a relational interpretation). The fourth rule is illegal since the lattice variable l occurs in a key position in the head of the rule.

Formally, the original version of Flix enforces the *lattice range restriction*:

► **Definition 23** (Lattice Range Restriction). *Every lattice variable must occur in a lattice position.*

To understand the lattice range restriction, let us revisit the example from Section 2.

► **Example 24.** Figure 2 contains the Datalog rule:

```
Edge(c1, c2) :- fix Component(_, c1), fix Component(_, c2).
```


Assume that the rule did not use `fix` and that we ignore the lattice range restriction. Suppose we have the following lattice facts:

```
Component(7; {1}).      Component(7; {2}).      Component(7, {7}).
Component(5, {3}).     Component(5, {5}).
```

The minimal model has two facts: `Component(7; {1, 2, 7})` and `Component(5; {3, 5})`. We would thus assume that the above rule would compute the undirected edge fact: `{1, 2, 7} ↔ {3, 5}`. But this is not what the program computes! It derives nonsensical facts such as `{1} ↔ {3}`, `{1} ↔ {3, 5}`, and all other edges between intermediate lattice values. The lattice range restriction avoids this problem by banning the program. We propose to allow the program as long as the lattice is `fix`'ed, i.e. fully computed, before it is used in a relation.

As the example shows, the lattice range restriction is overly strict. We can allow lattice variables to be used in key positions in head atoms provided that we ensure that the head predicate symbol occurs in a strictly higher stratum than every body atom in which the lattice variable occurs. This ensures that the lattice predicates are fully computed before they are used as keys. We introduce the extended dependency graph to capture this notion:

► **Definition 25** (Extended Dependency Graph). *The extended dependency graph of a Datalog program D with lattice semantics is a directed graph of predicate symbols that contains:*

- a weak edge $a \leftarrow b$ if D contains a rule where a is the predicate symbol of the head atom and b is a predicate symbol of a positive body atom, and
- a strong edge $a \leftarrow\!-\! b$ if D contains a rule where a is the predicate symbol of the head and b is a predicate symbol of a fixed or negative body atom.

We use the word *strong* to represent either a *fixed* or a *negative* dependency.

We also update the range restriction:

► **Definition 26** (Extended Lattice Range Restriction). *If every occurrence of a lattice variable is under a `fix` in the body of a rule, then the variable may be treated as a key in the head of the rule.*

Finally, we update our definition of stratification for the λ_{DAT} calculus and for Flix:

► **Definition 27** (Extended Stratification). *A Datalog program D_L , enriched with lattice semantics, is stratified if the extended dependency graph does not contain a cycle with a strong edge. A λ_{DAT} calculus program P_L , enriched with lattice semantics, is stratified if every Datalog value constructed during evaluation of P_L is stratified.*

► **Example 28.** We conclude with a small example of how the `fix` construct can be used¹:

```
Degree("Kevin Bacon"; Down(0)).
Degree(x; n + Down(1)) :- Degree(y; n), StarsWith(y, x).
Layer(n; Set#{x}) :- fix Degree(x; n).
Count(n, Set.size(s)) :- fix Layer(n; s).
```

This program computes the number of people who are separated by n -degrees from Kevin Bacon. For example, the 2nd-degree is *the number of people* who have starred in a movie where someone in that movie has also starred in another movie with Kevin Bacon.

¹ Here, the `Down` constructor defines a lattice with the reverse ordering of the underlying type.

7 Implementation

We now describe where the original version of Flix, and our proposed future version of Flix, reside in the design space.

7.1 The Original Flix Implementation

The original Flix implementation and its associated paper [30] do not use the terminology of this paper. Nevertheless, we can recast their design choices in our framework.

Flix uses a labelled dependency graph constructed from the entire program and filtered based on the type of each Datalog expression, the hybrid approach. The labels are predicate symbols where each dependency edge is annotated with its source and destination (**choice 1b**). No enriched labelling or types are used (**choice 2d**). Stratification is performed *without* monomorphization (**choice 3a**).

7.2 Our Flix Extension

While all design choices are valid, in our view, **choice 1b** and **choice 2d** are sub-optimal.

- For **choice 1b**, by only using the predicate symbols that are the source and destination of each dependency edge, we lose the important information that most dependency edges arise from rules where multiple predicate symbols are involved and thus where *all* of these predicate symbols must be present for the edge to be relevant. Instead, **choice 1c** or **choice 1d** offer increased precision with little downside.
- For **choice 2d**, ignoring the arity and term types of each predicate symbol loses important contextual information. In other logic programming languages, such as Prolog, predicate symbols are often overloaded and use the arity as part of their name, e.g. `spawn/2` and `spawn/3`. Given that λ_{DAT} and Flix are statically typed, it seems like a missed opportunity not to use types to distinguish predicate symbols, e.g. `Path(Int32, Int32)` vs. `Path(String, String)`.

For these reasons, in our proposed extension, we settled on **choice 1c** and **choice 2b** combined with **choice 2c**. We chose **choice 1c** because of its increased precision while still remaining explainable to the programmer. For **choice 2b** and **choice 2c**, we think that incorporating types and the distinction between relational and lattice predicates into the dependence graph increases precision significantly while also remaining understandable to the programmer.

We keep the rest of the design choices the same. We explored the idea of moving the stratifier after monomorphization **choice 3b**, which would boost precision. However, monomorphization is a relatively expensive compiler phase that is traditionally not part of the Flix compiler frontend. Thus, if stratification depends on monomorphization, then it becomes part of the frontend and must be run whenever the program is “type checked” by an IDE. We were worried that this would have unacceptable performance implications².

In summary, our Flix extension makes the following design choices:

- We use the hybrid approach based on the *labelled dependency graph*.
- **Choice 1c**: We use *rule-level* granularity.
- **Choice 2b**: We enrich the graph with *term types*.
- **Choice 2c**: We enrich the graph with *relation and lattice* information.
- **Choice 3a**: We stratify *without* monomorphization.

² This frontend versus backend problem is not unique to stratification. For example, many C or C++ compilers will report additional compilation warnings or errors when expensive backend optimizations are enabled. This might seem counter-intuitive, but the reason is that expensive program analysis enables the compiler to know more about the program and thus to report more warnings or errors.

7.3 Implementation Details

We have implemented the above design choices in an extension of the Flix compiler.

Flix is a functional-first, imperative, and logic programming language that supports algebraic data types, pattern matching, higher-order functions, parametric polymorphism, type classes, higher-kinded types, polymorphic effects, extensible records, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination [30, 31].

The Flix compiler project, including the standard library and tests, consists of 161,000 lines of Flix and Scala code. We re-wrote the Stratifier compiler phase which required ~1,000 lines of code and added support for the “fix” construct which required ~700 lines of code.

Flix, with our extensions, is ready for use, open source, and freely available at:

<https://flix.dev> and <https://github.com/flix/flix>

7.4 When a Program Does Not Stratify

When Flix programmers encounter a stratification error, there are essentially two possibilities:

- The program contains an actual stratification error.
- The type system is too imprecise to rule out the possibility of a stratification error.

We want to support the programmer in both scenarios. First, this means giving the programmer useful error messages such that they can accurately identify which of the two cases is applicable. Second, we want to give the programmer the ability to refactor their program such that it passes the stratification.

If a programmer should encounter a stratification error due to imprecision, they can:

1. Rename a predicate symbol to avoid a clash with a conceptually different predicate symbol. For example, instead of `Node`, a more suitable name could be `City`.
2. Introduce an extra predicate symbol in a rule to exclude the clash.
3. Change the arity of a predicate symbol, for example by recording more or less information.
4. Enrich the types of the terms in a predicate; for example, the programmer could introduce a new type `Celsius` instead of `Int32` or a new type `City` instead of `String`.

We think these are flexible and reasonable strategies that a Flix programmer will be able to use. Of course these strategies cannot necessarily resolve *all* stratification issues, but the space of accepted programs is much larger than in the original version of Flix where only strategy (1) is available.

7.5 The Motivating Example, Revisited

We now revisit the motivating example from Section 2. Figure 6 shows the labelled dependency graph for the program in Figure 2. The graph reflects our design choices:

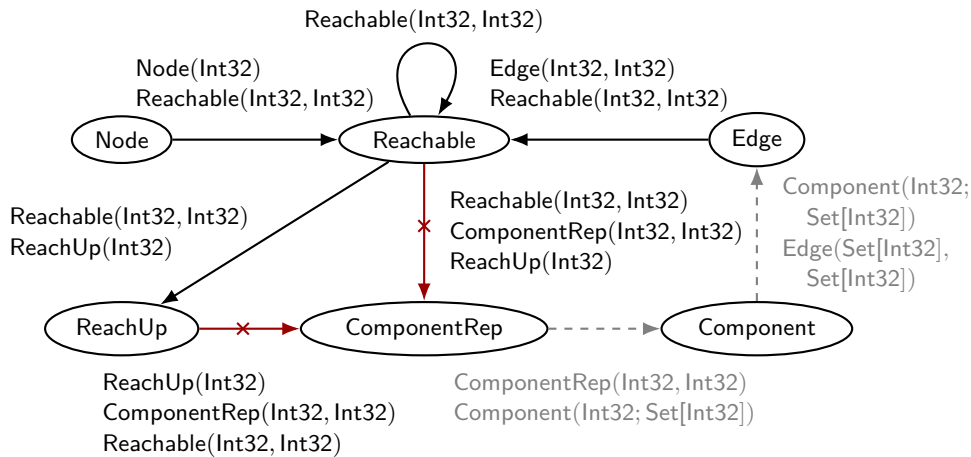
- (i) the dependency edges are labelled with all predicate symbols from the rule that gives rise to the dependency, and
- (ii) the labels carry the term types of each predicate symbol.

The braces and commas of the label sets are omitted. Inspecting the graph, we see two potential negative cycles. However, no expression in the program in Figure 2 has a type where the cycles cannot be excluded.

As an example, on line 26, the composition of the expressions `ns`, `es`, and `r` has the type:

```
{ComponentRep = (Int32, Int32), Reachable = (Int32, Int32),
  Node = (Int32), Edge = (Int32, Int32), ReachUp = (Int32)}
```

31:22 Breaking the Negative Cycle



■ **Figure 6** The labelled dependency graph of the program in Figure 2 based on the design decisions of Section 7.2. The gray dashed lines are the edges that are filtered out when looking at the union of `ns`, `es`, `r` on line 26.

This excludes the dashed dependency edges shown in gray in Figure 6. For example, the edge between `Component` and `Edge` is ruled out since the term types of `Edge` do not match and since `Component` is not present. The resulting graph, shown with solid arrows, does not contain any negative cycles. Hence the expression is stratified. This is true for all expressions in the program, hence Flix is able to statically determine that the program is stratified.

8 Case Study: A Small Graph Library in Flix

We have implemented a small open-source graph library for Flix³. The graph library provides a range of queries on graphs, as shown in Table 1. Each query is implemented as a Flix function that internally uses Datalog. The purpose of the case study is to explore how a graph library can be implemented using first-class Datalog constraints in Flix and to check that the proposed stratification strategy is practical.

Table 1 shows an overview of the functions that we have implemented in the graph library. In total, we have implemented 24 functions in 450 lines of code. The columns of the table are as follows: The `Function` and `Lines` columns give the name and number of source code lines for a specific function. The `Lattices` column indicates whether the Datalog program uses lattice semantics. The `Stratified Negation` column indicates whether the Datalog program uses stratified negation. The `Fix Construct` column indicates whether the Datalog program uses the fix construct.

To look at a specific example, the table shows that the `stronglyConnectedComponents` function consists of 26 lines of code, it uses lattice semantics and the fix construct, but it does not use stratified negation. To give an idea of how the library is implemented, the complete source code for the `stronglyConnectedComponents` function is shown in Figure 7. As the code shows, each query is implemented using first-class Datalog constraints and there is some code-reuse in the form of the `nodes` and `reachability` functions which return Datalog program values.

³ <https://github.com/flix/flix/blob/master/main/src/library/Graph.flix>

■ **Table 1** Overview of the Flix Graph Library. The line count includes the number of lines of helper functions. The features used in a function also include features used in helper functions.

Function	Lines	Features Used		
		Lattices	Stratified Negation	Fix Construct
boundedDistances	20	Y	N	Y
closure	17	N	N	N
cutPoints	30	Y	Y	Y
degrees	19	Y	N	Y
distance	16	Y	N	Y
distances	49	Y	N	Y
distancesFrom	15	Y	N	Y
flipEdges	10	N	N	N
frontiersFrom	29	Y	N	Y
inDegrees	18	Y	N	Y
invert	12	N	Y	N
isCyclic	15	N	N	N
outDegrees	18	Y	N	Y
reachable	16	N	N	N
reachableFrom	15	N	N	N
stronglyConnectedComponents	26	Y	N	Y
toGraphviz	11	N	N	N
toGraphvizWeighted	11	N	N	N
topologicalsort	28	Y	Y	N
toUndirected	4	N	N	N
toUndirectedWeighted	4	N	N	N
unreachableFrom	20	N	Y	N
withinDistanceOf	14	Y	N	Y
withinEdgesOf	14	Y	N	Y

In total, the library contains 31 distinct predicate symbols and 64 rules. The library was developed using our extended Flix compiler. During development, we never encountered a spurious stratification error. However, if we compile the library with the original Flix compiler, it is unfairly rejected due to a spurious negative cycle.

In summary, Table 1 shows that: (i) we have 24 functions that co-exist, using overlapping predicate names from the same domain, without spurious stratification errors, (ii) the majority of functions (15/24) require stratification via `not`, `fix`, or both, (iii) many functions use lattice semantics (13/24), (iv) the `fix` construct is used more often than the `not` construct, and (v) the library is accepted by our extended Flix compiler, but is rejected by the original Flix compiler due to a spurious negative cycle in the dependency graph.

Flix is a whole-program optimizing compiler. When the graph library is compiled together with the standard library, the stratification is computed in 0.16 seconds whereas the total compilation time is 7.3 seconds. In particular, compilation time is dominated by the cost of type inference. In conclusion, we find the stratification does not unfairly reject our library and that the cost of computing the stratification is low.

31:24 Breaking the Negative Cycle

```
1 /// Returns the strongly connected components of the directed
2 /// graph 'g'. Two nodes are in the same component if and only
3 /// if they can both reach each other.
4 pub def stronglyConnectedComponents(g: m[(t, t)]): Set[Set[t]]
5   with Foldable[m], Boxable[t] = {
6   let edges = inject g into Edge;
7   let connected = #{
8     // If 'n1' can reach 'n2' and 'n2' can reach 'n1' then they are
9     // part of the same strongly connected component.
10    Connected(n1; Set#{n2}) :- Reachable(n1, n2), Reachable(n2, n1).
11  };
12  let components = #{
13    // After the full computation of 'Connected', duplicates are
14    // removed by checking that 'n' is the minimum in the strongly
15    // connected component.
16    Components(s) :- fix Connected(n; s), if Some(n) == Set.minimum(s).
17  };
18  let res = query edges, nodes(), reachability(), connected, components
19    select x
20    from Components(x);
21  List.toSet(res)
22 }
```

■ **Figure 7** The `stronglyConnectedComponents` function from the graph library case study.

9 Related Work

9.1 First-class Datalog

Magnus and Lhoták present the λ_{DAT} calculus which is the foundation for the current work [30]. The authors briefly discuss stratified negation and propose a simple solution based on type filtering similar to **choice 1b** without any information on the labels (**choice 2d**). As we have seen, some of these choices are sub-optimal.

9.2 Negation and Aggregation Semantics

There are many proposed semantics for Datalog with negation but stratified negation is the most prevalent one. Kolaitis and Papadimitriou present inflationary semantics that produce facts in such a way that a fixpoint exists for all programs using negation. The fixpoint is not guaranteed to be minimal [26]. There are also variations in the realm of stratification. Negation can be restricted to guarded negation, which in broad terms means that all first-order variables in negated atoms exist in a single atom. This makes additional questions like query containment decidable [5]. Local stratification stratifies the program on instances of rules instead of the quantified rules. This is a property that is hard to verify, but in its most expressive form, it allows deducing $\text{even}(i + 1)$ from $\neg\text{even}(i)$, since this is not circular reasoning for any instantiations of i [36].

Aggregation is non-monotone like negation, which is why it has been studied using many of the same ideas. Aggregation can naturally be stratified like negation but another option is group-stratification based on the standard group-by operation. This means that a group of the predicate should not depend on itself [35]. Zaniolo et al. studies both negation and aggregation with a syntactically restricted form of local stratification that essentially tracks the dynamic strata on the facts [48].

9.3 Datalog Extensions

There have been many efforts to increase the expressive power and usability of Datalog while maintaining practical feasibility. One extension is the existential quantification of variables in the rule head. This was motivated by the ontological reasoning that is needed in web-standards for databases [19]. Datalog[∃] is undecidable, so a family of languages called Datalog[±] [8] makes restrictions that reduce the complexity to classes ranging from AC₀ to EXPTIME. One of these is Warded Datalog[±] [19], which has a syntactic restriction on the usage of variables that may be bound to non-constant variables in evaluation and flow into the rule head. They must be within a single predicate, the “ward”. It uses stratified negation and negative constraints that restrict the inclusion of certain facts.

10 Conclusion

Flix is a functional, imperative, and logic language with support for first-class Datalog constraints. In Flix, Datalog constraints are values that can be constructed, passed as arguments to functions, returned from functions, composed with other Datalog values, and solved. Flix is based on the λ_{DAT} calculus which itself builds on the Hindley-Milner type system extended with row polymorphism. A significant part of the expressive power of Datalog comes from the use of negation. Stratified negation is a particular simple form of negation that prohibits recursion through negation and is easily accessible to ordinary programmers. While it is straightforward to determine if a Datalog program is stratified, it is much more difficult to statically determine if a λ_{DAT} program is stratified. In this paper, we have explored the design space of stratification for λ_{DAT} . We have proposed several improvements to stratification in Flix and we have implemented these. With our extension, Flix accepts a much broader range of programs that use stratified negation. Finally, we have also extended Flix with a new “fix” construct that enables lattice values to be used as relational values.

References

- 1 Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. Dedalus: Datalog in time and space. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 262–281, Berlin, 2011. Springer Berlin Heidelberg.
- 2 Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The disjunctive Datalog system DLV. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 282–301, Berlin, 2011. Springer Berlin Heidelberg.
- 3 Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proc. of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15, pages 1371–1382, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2723372.2742796.
- 4 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proc. in Informatics (LIPIcs)*, pages 2:1–2:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.2.
- 5 Vince Bárány, Balder ten Cate, and Martin Otto. Queries with guarded negation. *Proc. VLDB Endow.*, 5(11):1328–1339, July 2012. doi:10.14778/2350229.2350250.

- 6 Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for SMT-based static analysis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428209.
- 7 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1640089.1640108.
- 8 Andrea Cali, Georg Gottlob, and Thomas Lukasiewicz. A general Datalog-based framework for tractable query answering over ontologies. In *Proc. of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '09, pages 77–86, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1559795.1559809.
- 9 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases: An Overview*, pages 1–15. Springer Berlin Heidelberg, Berlin, 1990. doi:10.1007/978-3-642-83952-8_1.
- 10 Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proc. of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2391229.2391230.
- 11 Luis Damas. *Type assignment in programming languages*. PhD thesis, The University of Edinburgh, 1984.
- 12 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/582153.582176.
- 13 Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Trans. Database Syst.*, 22(3):364–418, September 1997. doi:10.1145/261124.261126.
- 14 Melvin Fitting. Fixpoint semantics for logic programming a survey. *Theoretical Computer Science*, 278(1):25–51, 2002. Mathematical Foundations of Programming Semantics 1996. doi:10.1016/S0304-3975(00)00330-3.
- 15 Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1075–1092, Berkeley, California, USA, August 2020. USENIX Association.
- 16 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski, Bowen, and Kenneth, editors, *Proc. of International Logic Programming Conference and Symposium*, pages 1070–1080, Cambridge, MA, USA, 1988. MIT Press.
- 17 Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3):365–385, August 1991. doi:10.1007/BF03037169.
- 18 G. Gottlob, S. Ceri, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(01):146–166, 1989. doi:10.1109/69.43410.
- 19 Georg Gottlob and Andreas Pieris. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *Proc. of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 2999–3007, Palo Alto, California, USA, 2015. AAAI Press.
- 20 Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspoul Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the Myria big data management service. In *Proc. of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2588555.2594530.
- 21 R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. doi:10.2307/1995158.

- 22 Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.
- 23 Mark Kaminski, Bernardo Cuenca Grau, Egor V. Kostylev, Boris Motik, and Ian Horrocks. Foundations of declarative data analysis using limit Datalog programs. In *Proc. of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1123–1130, California, USA, 2017. International Joint Conferences on Artificial Intelligence. doi:10.24963/ijcai.2017/156.
- 24 Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. Distribution policies for Datalog. In Benny Kimelfeld and Yael Amsterdamer, editors, *21st International Conference on Database Theory (ICDT 2018)*, volume 98 of *Leibniz International Proc. in Informatics (LIPIcs)*, pages 17:1–17:22, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ICDT.2018.17.
- 25 Ross D. King. Applying inductive logic programming to predicting gene function. *AI Mag.*, 25(1):57–68, March 2004.
- 26 Phokion G. Kolaitis and Christos H. Papadimitriou. Why not negation by fixpoint? In *Proc. of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '88, pages 231–239, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/308386.308446.
- 27 Kenneth Kunen. Negation in logic programming. *The Journal of Logic Programming*, 4(4):289–308, 1987. doi:10.1016/0743-1066(87)90007-0.
- 28 Daan Leijen. Extensible records with scoped labels. *Trends in Functional Programming*, 5:297–312, 2005.
- 29 Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, November 2009. doi:10.1145/1592761.1592785.
- 30 Magnus Madsen and Ondřej Lhoták. Fixpoints for the masses: Programming with first-class Datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428193.
- 31 Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From Datalog to Flix: A declarative language for fixed points on lattices. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, PLDI '16, pages 194–208, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2908080.2908096.
- 32 Hongyuan Mei, Guanghui Qin, Minjie Xu, and Jason Eisner. Neural Datalog through time: Informed temporal modeling via logical specification. In *International Conference on Machine Learning*, pages 6808–6819, Madison, WI, USA, 2020. PMLR, Omnipress.
- 33 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. doi:10.1016/0022-0000(78)90014-4.
- 34 Raymond J. Mooney. Inductive logic programming for natural language processing. In Stephen Muggleton, editor, *Inductive Logic Programming*, pages 1–22, Berlin, 1997. Springer Berlin Heidelberg.
- 35 Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proc. of the Sixteenth International Conference on Very Large Databases*, pages 264–277, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
- 36 Teodor C. Przymusiński. Chapter 5 - on the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, USA, 1988. doi:10.1016/B978-0-934613-40-8.50009-9.
- 37 Kenneth A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proc. of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 161–171, New York, NY, USA, 1990. Association for Computing Machinery. doi:10.1145/298514.298558.

- 38 Jiwon Seo. Datalog extensions for bioinformatic data analysis. *Annu Int Conf IEEE Eng Med Biol Soc*, 2018:1303–1306, July 2018.
- 39 Jiwon Seo, Stephen Guo, and Monica S. Lam. SocialLite: Datalog extensions for efficient social network analysis. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 278–289, Manhattan, New York, USA, 2013. IEEE. doi:10.1109/ICDE.2013.6544832.
- 40 Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed Socialite: A Datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, September 2013. doi:10.14778/2556549.2556572.
- 41 Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with Datalog queries on Spark. In *Proc. of the 2016 International Conference on Management of Data, SIGMOD '16*, pages 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2882903.2915229.
- 42 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, pages 245–251, Berlin, 2011. Springer Berlin Heidelberg.
- 43 Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276509.
- 44 Petar Tsankov. Security analysis of smart contracts in Datalog. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 316–322, Cham, 2018. Springer International Publishing.
- 45 Jeffrey D. Ullman. Principles of database and knowledge-base systems, 1988.
- 46 Mario Wenzel and Stefan Brass. Declarative programming for microcontrollers - Datalog on Arduino. In *Declarative Programming and Knowledge Management: Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9–12, 2019, Revised Selected Papers*, pages 119–138, Berlin, 2019. Springer-Verlag. doi:10.1007/978-3-030-46714-2_9.
- 47 Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.
- 48 Carlo Zaniolo, Natraj Arni, and Kayliang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In Stefano Ceri, Katsumi Tanaka, and Shalom Tsur, editors, *Deductive and Object-Oriented Databases*, pages 204–221, Berlin, 1993. Springer Berlin Heidelberg.

Asynchronous Multiparty Session Type Implementability is Decidable – Lessons Learned from Message Sequence Charts

Felix Stutz  

MPI-SWS, Kaiserslautern, Germany

Abstract

Multiparty session types (MSTs) provide efficient means to specify and verify asynchronous message-passing systems. For a global type, which specifies all interactions between roles in a system, the implementability problem asks whether there are local specifications for all roles such that their composition is deadlock-free and generates precisely the specified executions. Decidability of the implementability problem is an open question. We answer it positively for global types with sender-driven choice, which allow a sender to send to different receivers upon branching and a receiver to receive from different senders. To achieve this, we generalise results from the domain of high-level message sequence charts (HMSCs). This connection also allows us to comprehensively investigate how HMSC techniques can be adapted to the MST setting. This comprises techniques to make the problem algorithmically more tractable as well as a variant of implementability that may open new design space for MSTs. Inspired by potential performance benefits, we introduce a generalisation of the implementability problem that we, unfortunately, prove to be undecidable.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Multiparty session types, Verification, Message sequence charts

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.32

Related Version *Extended Version*: <https://arxiv.org/abs/2302.11272>

Funding This research was sponsored in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248 – CPEC.

Acknowledgements The author thanks Damien Zufferey, Emanuele D’Osualdo, Ashwani Anand, Rupak Majumdar, and the anonymous reviewers for their valuable feedback.

1 Introduction

Distributed message-passing systems are omnipresent and, therefore, designing and implementing them correctly is very important. However, this is a very difficult task at the same time. In fact, it is well-known that verifying such systems is algorithmically undecidable in general due to the combination of asynchrony (messages are buffered) and concurrency [13].

Multiparty Session Type (MST) frameworks [37, 38] provide efficient means to specify and verify such distributed message-passing systems (e.g., see the survey [5]). They have also been applied to various other domains like cyber-physical systems [46], timed systems [9], web services [65], and smart contracts [27]. In MST frameworks, global types are global specifications, which comprise all interactions between roles in a protocol. From a design perspective, it makes sense to start with such a global protocol specification – instead of a system with arbitrary communication between roles and a specification to satisfy.

Let us consider a variant of the well-known two buyer protocol from the MST literature, e.g., [54, Fig. 4(2)]. Two buyers *a* and *b* purchase a sequence of items from seller *s*. We informally describe the protocol and *emphasise* the interactions. At the start and after every purchase (attempt), buyer *a* can decide whether to buy the next item or whether they are *done*. For each item, buyer *a* *queries* its price and the seller *s* replies with the *price*.



© Felix Stutz;

licensed under Creative Commons License CC-BY 4.0

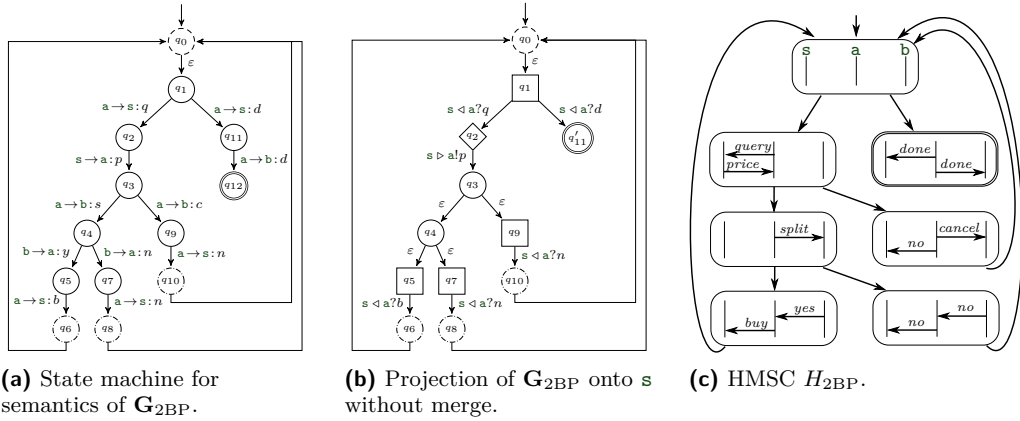
37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 32; pp. 32:1–32:31

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Two Buyer Protocol: the finite state machine for the semantics of G_{2BP} on the left, the first step of projection in the middle, and as HMSC on the right; a transition label $a \rightarrow s : q$ jointly specifies a send event $a \triangleright s ! q$ for buyer a and a receive event $s \triangleleft a ? q$ for seller s ; styles of states indicate their kind, e.g., recursion states (dashed lines) while final states have double lines.

Subsequently, buyer a decides whether to *cancel* the purchase process for the current item or proposes to *split* to buyer b that can *accept* or *reject*. In both cases, buyer a notifies the seller s if they want to *buy* the item or *not*. This protocol is specified with the following global type:

$$G_{2BP} := \mu t. + \left\{ \begin{array}{l} a \rightarrow s : query. s \rightarrow a : price. + \left\{ \begin{array}{l} a \rightarrow b : split. (b \rightarrow a : yes. a \rightarrow s : buy. t + b \rightarrow a : no. a \rightarrow s : no. t) \\ a \rightarrow b : cancel. a \rightarrow s : no. t \end{array} \right. \\ a \rightarrow s : done. a \rightarrow b : done. 0 \end{array} \right.$$

The first term μt binds the recursion variable t which is used at the end of the first two lines and allows the protocol to recurse back to this point. Subsequently, $+$ and the curly bracket indicate a choice by buyer a as it is the sender for the next interaction, e.g., $a \rightarrow s : query$. For our asynchronous setting, this term jointly specifies the send event $a \triangleright s ! query$ for buyer a and its corresponding receive event $s \triangleleft a ? query$ for seller s , which may happen with arbitrary delay. The state machine in Figure 1a illustrates its semantics with abbreviated message labels.

The Implementability Problem for Global Types and the MST Approach

A global type provides a global view of the intended protocol. However, when implementing a protocol in a distributed setting, one needs a local specification for each role. The *implementability problem* for a global type asks whether there are local specifications for all roles such that, when complying with their local specifications, their composition never gets stuck and exposes the same executions as specified by the global type. This is a challenging problem because roles can only partially observe the execution of a system: each role only knows the messages it sent and received and, in an asynchronous setting, a role does not know when one of its messages will be received by another role. In contrast, in a synchronous setting, there are no channels, yielding finite state systems. Still, we could not find a reference that precisely settles the decidability of synchronous implementability. We sketch a proof in Section 7. In this work, we solely deal with the asynchronous setting.

In general, one distinguishes between a role in a protocol and the process which implements the local specification of a role in a system. We use the local specifications directly as implementations so the difference is not essential and we use the term role instead of process.

Classical MST frameworks employ a partial *projection operator* with an in-built *merge operator* to solve the implementability problem. For each role, the projection operator takes the global type and removes all interactions the role is not involved in. Figure 1a illustrates the semantics of \mathbf{G}_{2BP} while Figure 1b gives the projection onto seller s before the merge operator is applied – in both, messages are abbreviated with their first letter. It is easy to see that this procedure introduces non-determinism, e.g., in q_3 and q_4 , which shall be resolved by the merge operator. Most merge operators can resolve the non-determinism in Figure 1b. A merge operator checks whether it is safe to merge the states and it might fail so it is a partial operation. For instance, every kind of state, indicated by a state’s style in Figure 1b, can only be merged with states of the same kind or states of circular shape. For a role, the result of the projection, if defined, is a local type. They act as local specifications and their syntax is similar to the one of global types.

Classical projection operators are a best-effort technique. This yields good (mostly linear) worst-case complexity but comes at the price of rejecting implementable global types. Intuitively, classical projection operators consider a limited search space for local types. They bail out early when encountering difficulties and do not unfold recursion. In addition, most MST frameworks do effectively not allow a role to send to different receivers or receive from different senders upon branching. This restriction is called *directed choice* – in contrast to *sender-driven choice* which is more permissive and allows these patterns. Among the classical projection operators, the one by Majumdar et al. [45] is the only to handle global types with sender-driven choice but it suffers from the shortcomings of a classical projection approach. We define different merge operators from the literature and visually explain their supported features by example. We show that the presented projection/merge operators fail to project implementable variations of the two buyer protocol, showcasing the sources of incompleteness for the classical approach. For non-classical approaches, we refer to Section 7.

As a best-effort technique, it is natural to focus on efficiency rather than completeness. The work by Castagna et al. [16] is a notable exception. Their notion of completeness [16, Def. 4.1] is not as strict as the one considered in this work and only a restricted version of their characterisation is algorithmically checkable. In general, it is not known whether the implementability problem for global types, with directed or sender-driven choice, is decidable. We answer this open question positively for global types with sender-driven choice. To this end, we relate the implementability problem for global types with the safe realisability problem for high-level message sequence charts and generalise results for the latter.

Lessons Learned from Message Sequence Charts

The two buyer protocol \mathbf{G}_{2BP} can also be specified as high-level message sequence chart (HMSC) [47], as illustrated in Figure 1c. Each block is a basic message sequence chart (BMSC) which intuitively corresponds to straight-line code. In each of those, time flows from top to bottom and each role is represented by a vertical line. We only give the names in the initial block, which is marked by an incoming arrow at the top. An arrow between two role lines specifies sending and receiving a message with its label. The graph structure adds branching, which corresponds to choice in global types, and control flow. Top branches from the global type are on the left in the HMSC while bottom branches are on the right.

While research on MSTs and HMSCs has been pursued quite independently, the MST literature frequently uses HMSC-like visualisations for global types, e.g., [15, Fig. 1] and [38, Figs. 1 and 2]. The first formal connection was recently established by Stutz and Zufferey [57].

The HMSC approach to the implementability problem, studied as safe realisability, differs from the MST approach of checking conditions during the projection. For an HMSC, it is known that there is a candidate implementation [2], which implements the HMSC if it is implementable. Intuitively, one takes the HMSC and removes all interactions a role is not involved in and determinises the result. We generalise this result to infinite executions.¹

Hence, algorithms and conditions center around checking implementability of HMSCs. In general, this problem is undecidable [44]. For *globally-cooperative* HMSCs [31], Lohrey [44] proved it to be EXPSPACE-complete. We show that any implementable global type belongs to this class of HMSCs.¹ These results give rise to the following algorithm to check implementability of a global type. One can check whether a global type is globally-cooperative (which is equivalent to checking its HMSC encoding). If it is not globally-cooperative, it cannot be implementable. If it is globally-cooperative, we apply the algorithm by Lohrey [44] to check whether its HMSC encoding is implementable. If it is, we use its candidate implementation and know that it generalises to infinite executions.

While this algorithm shows decidability, the complexity might not be tractable. Based on our results, we show how more tractable but still permissive approaches to check implementability of HMSCs can be adapted to the MST setting. In addition, we consider *payload implementability*, which allows to add payload to messages of existing interactions and checks agreement when the additional payload is ignored. We present a sufficient condition for global types that implies payload implementability. These techniques can be used if the previous algorithms are not tractable or reject a global type.

Furthermore, we introduce a generalisation of the implementability problem. A network may reorder messages from different senders for the same receiver but the implementability problem still requires the receiver to receive them in the specified order. Our generalisation allows to consider such reorderings of arrival and can yield performance gains. In addition, it also renders global types implementable that are not implementable in the standard setting. Unfortunately, we prove this generalisation to be undecidable in general.

Contributions and Outline

We introduce our MST framework in Section 2 while Section 7 covers related work. In the other sections, we introduce the necessary concepts to establish our main *contributions*:

- We give a visual explanation of the classical projection operator with different merge operators and exemplify its shortcomings (Section 3).
- We prove decidability of the implementability problem for global types with sender-driven choice (Section 4) – provided that protocols can (but do not need to) terminate.
- We comprehensively investigate how MSC techniques can be applied to the MST setting, including algorithmics with better complexity for subclasses as well as an interesting variant of the implementability problem (Section 5).
- Lastly, we introduce a new variant of the implementability problem with a more relaxed role message ordering, which is closer to the network ordering, and prove it to be undecidable in general (Section 6).

¹ For this, we impose a mild assumption: all protocols can (but do not need to) terminate.

2 Multiparty Session Types

In this section, we formally introduce our Multiparty Session Type (MST) framework. We define the syntax of global and local types and their semantics. Subsequently, we recall the implementability problem for global types which asks if there is a deadlock-free communicating state machine that admits the same language (without additional synchronisation).

Finite and Infinite Words. Let Σ be an alphabet. We denote the set of finite words over Σ by Σ^* and the set of infinite words by Σ^ω . Their union is denoted by Σ^∞ . For two strings $u \in \Sigma^*$ and $v \in \Sigma^\infty$, we say that u is a *prefix* of v if there is some $w \in \Sigma^\infty$ such that $u \cdot w = v$ and denote this with $u \leq v$ while $\text{pref}(v)$ denotes all prefixes of v and is lifted to languages as expected. For a language $L \subseteq \Sigma^\infty$, we distinguish between the language of finite words $L_{\text{fin}} := L \cap \Sigma^*$ and the language of infinite words $L_{\text{inf}} := L \cap \Sigma^\omega$.

Message Alphabet. We fix a finite set of messages \mathcal{V} and a finite set of roles \mathcal{P} , ranged over with p, q, r , and s . With $\Sigma_{\text{sync}} = \{p \rightarrow q : m \mid p, q \in \mathcal{P} \text{ and } m \in \mathcal{V}\}$, we denote the set of interactions where sending and receiving a message is specified at the same time. For our asynchronous setting, we also define individual send and receive events: $\Sigma_p = \{p \triangleright q!m, p \triangleleft q?m \mid q \in \mathcal{P}, m \in \mathcal{V}\}$ for a role p . For both send events $p \triangleright q!m$ and receive events $p \triangleleft q?m$, the first role is *active*, i.e., the sender in the first event and the receiver in the second one. The union for all roles yields all (asynchronous) events: $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$. For the rest of this work, we fix the set of roles \mathcal{P} , the messages \mathcal{V} , and both sets Σ_{sync} and Σ . We may also use the term Σ_{async} for Σ . We define an operator that splits events from Σ_{sync} , $\text{split}(p \rightarrow q : m) := p \triangleright q!m. q \triangleleft p?m$, which is lifted to sequences and languages as expected. Given a word, we might also project it to all letters of a certain shape. For instance, $w \downarrow_{p \triangleright q!}$ is the subsequence of w with all of its send events where p sends any message to q . If we want to select all messages of w , we write $\mathcal{V}(w)$.

Global and Local Types – Syntax

We give the syntax of global and local types following work by Majumdar et al. [45]. In this work, we consider global types as specifications for message-passing concurrency and omit features like delegation.

► **Definition 2.1** (Syntax of global types). Global types for MSTs are defined by the grammar:

$$G ::= 0 \mid \sum_{i \in I} p \rightarrow q_i : m_i . G_i \mid \mu t . G \mid t$$

The term 0 explicitly represents termination. A term $p \rightarrow q_i : m_i$ indicates an interaction where p sends message m_i to q_i . In our asynchronous semantics, it is split into a send event $p \triangleright q_i!m_i$ and a receive event $q_i \triangleleft p?m_i$. In a choice $\sum_{i \in I} p \rightarrow q_i : m_i . G_i$, the sender p chooses the branch. We require choices to be unique, i.e., $\forall i, j \in I. i \neq j \Rightarrow q_i \neq q_j \vee m_i \neq m_j$. If $|I| = 1$, which means there is no actual choice, we omit the sum operator. The operators μt and t allow to encode loops. We require them to be guarded, i.e., there must be at least one interaction between the binding μt and the use of the recursion variable t . Without loss of generality, all occurrences of recursion variables t are bound and distinct.

Our global types admit *sender-driven choice* as p can send to different receivers upon branching: $\sum_{i \in I} p \rightarrow q_i : m_i . G_i$. This is also called generalised choice by Majumdar et al. [45]. In contrast, *directed choice* requires a sender to send to a single receiver, i.e., $\forall i, j \in I. q_i = q_j$.

► **Example 2.2** (Global types). The two buyer protocol \mathbf{G}_{2BP} from the introduction is a global type. Instead of \sum , we use $+$ with curly brackets.

► **Definition 2.3** (Syntax of local types). For a role p , the local types are defined as follows:

$$L ::= 0 \mid \bigoplus_{i \in I} q_i!m_i.L_i \mid \&_{i \in I} q_i?m_i.L_i \mid \mu t.L \mid t$$

We call $\bigoplus_{i \in I} q_i!m_i$ an internal choice while $\&_{i \in I} q_i?m_i$ is an external choice. For both, we require the choice to be unique, i.e., $\forall i, j \in I. i \neq j \Rightarrow q_i \neq q_j \vee m_i \neq m_j$. Similarly to global types, we may omit \bigoplus or $\&$ if there is no actual choice and we require recursion to be guarded as well as recursion variables to be bound and distinct.

► **Example 2.4** (Local type). For the global type \mathbf{G}_{2BP} , a local type for seller s is

$$\mu t. \& \left\{ \begin{array}{l} a?query. a!price. (a?buy. t \& a?no. t) \\ a?done. 0 \end{array} \right.$$

Implementing in a Distributed Setting

Global types can be thought of as global protocol specifications. Thus, a natural question and a main concern in MST theory is whether a global type can be implemented in a distributed setting. We present communicating state machines, which are built from finite state machines, as the standard implementation model.

► **Definition 2.5** (State machines). A state machine $A = (Q, \Delta, \delta, q_0, F)$ is a 5-tuple with a finite set of states Q , an alphabet Δ , a transition relation $\delta \subseteq Q \times (\Delta \cup \{\varepsilon\}) \times Q$, an initial state $q_0 \in Q$ from the set of states, and a set of final states F with $F \subseteq Q$. If $(q, a, q') \in \delta$, we also write $q \xrightarrow{a} q'$. A sequence $q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \dots$, with $q_i \in Q$ and $w_i \in \Delta \cup \{\varepsilon\}$ for $i \geq 0$, such that q_0 is the initial state, and for each $i \geq 0$, it holds that $(q_i, w_i, q_{i+1}) \in \delta$, is called a run in A with its trace $w_0 w_1 \dots \in \Delta^\infty$. A run is maximal if it ends in a final state or is infinite. The language $\mathcal{L}(A)$ of A is the set of traces of all maximal runs. If Q is finite, we say A is a finite state machine (FSM).

► **Definition 2.6** (Communicating state machines). We call $\{\{A_p\}\}_{p \in \mathcal{P}}$ a communicating state machine (CSM) over \mathcal{P} and \mathcal{V} if A_p is a finite state machine with alphabet Σ_p for every $p \in \mathcal{P}$. The state machine for p is denoted by $(Q_p, \Sigma_p, \delta_p, q_{0,p}, F_p)$. Intuitively, a CSM allows a set of state machines, one for each role in \mathcal{P} , to communicate by sending and receiving messages. For this, each pair of roles $p, q \in \mathcal{P}$, $p \neq q$, is connected by two directed message channels. A transition $q_p \xrightarrow{p \triangleright q!m} q'_p$ in the state machine of p denotes that p sends message m to q if p is in the state q_p and changes its local state to q'_p . The channel $\langle p, q \rangle$ is appended by message m . For receptions, a transition $q_q \xrightarrow{q \triangleleft p?m} q'_q$ in the state machine of q corresponds to q retrieving the message m from the head of the channel when its local state is q_q which is updated to q'_q . The run of a CSM always starts with empty channels and each finite state machine is in its respective initial state. A deadlock of $\{\{A_p\}\}_{p \in \mathcal{P}}$ is a reachable configuration without outgoing transitions such that there is a non-empty channel or some participant is in a non-final local state. The formalisation of this intuition is standard and can be found in the technical report [56].

A global type always specifies send and receive events together. In a CSM execution, there may be independent events that can occur between a send and its respective receive event.

► **Example 2.7** (Motivation for indistinguishability relation \sim). Let us consider the following global type which is a part of the two buyer protocol: $a \triangleright b : cancel. a \triangleright s : no. 0$. This is one of its traces: $a \triangleright b!cancel. b \triangleleft a?cancel. a \triangleright s!no. s \triangleleft a?no$. Because the active roles in

$b \triangleleft a?cancel$ and $a \triangleright s!no$ are different and we do not reorder a receive event in front of its respective send event, any CSM that accepts the previous trace also accepts the following trace: $a \triangleright b!cancel. a \triangleright s!no. b \triangleleft a?cancel. s \triangleleft a?no$.

Majumdar et al. [45] introduced the following relation to capture this phenomenon.

► **Definition 2.8** (Indistinguishability relation \sim [45]). *We define a family of indistinguishability relations $\sim_i \subseteq \Sigma^* \times \Sigma^*$, for $i \geq 0$. For $w \in \Sigma^*$, we have $w \sim_0 w$. For $i = 1$, we define:*

1. *If $p \neq r$, then $w.p \triangleright q!m.r \triangleright s!m'.u \sim_1 w.r \triangleright s!m'.p \triangleright q!m.u$.*
2. *If $q \neq s$, then $w.q \triangleleft p?m.s \triangleleft r?m'.u \sim_1 w.s \triangleleft r?m'.q \triangleleft p?m.u$.*
3. *If $p \neq s \wedge (p \neq r \vee q \neq s)$, then $w.p \triangleright q!m.s \triangleleft r?m'.u \sim_1 w.s \triangleleft r?m'.p \triangleright q!m.u$.*
4. *If $|w \downarrow_{p \triangleright q!}| > |w \downarrow_{q \triangleleft p?}|$, then $w.p \triangleright q!m.q \triangleleft p?m'.u \sim_1 w.q \triangleleft p?m'.p \triangleright q!m.u$.*

Let w, w' , and w'' be words s.t. $w \sim_1 w'$ and $w' \sim_i w''$ for some i . Then, $w \sim_{i+1} w''$. We define $w \sim u$ if $w \sim_n u$ for some n . It is straightforward that \sim is an equivalence relation. Define $u \preceq v$ if there is $w \in \Sigma^$ such that $u.w \sim v$. Observe that $u \sim v$ iff $u \preceq v$ and $v \preceq u$. For infinite words $u, v \in \Sigma^\omega$, we define $u \preceq^\omega v$ if for each finite prefix u' of u , there is a finite prefix v' of v such that $u' \preceq v'$. Define $u \sim v$ iff $u \preceq^\omega v$ and $v \preceq^\omega u$.*

We lift the equivalence relation \sim on words to languages:

For a language L , we define $\mathcal{C}^\sim(L) = \left\{ w' \mid \bigvee \begin{array}{l} w' \in \Sigma^ \wedge \exists w \in \Sigma^*. w \in L \text{ and } w' \sim w \\ w' \in \Sigma^\omega \wedge \exists w \in \Sigma^\omega. w \in L \text{ and } w' \preceq^\omega w \end{array} \right\}$.*

This relation characterises what can be achieved in a distributed setting using CSMs.

► **Lemma 2.9** (L. 21 [45]). *Let $\{\{A_p\}_{p \in \mathcal{P}}\}$ be a CSM. Then, $\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}) = \mathcal{C}^\sim(\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}))$.*

Global and Local Types – Semantics

Hence, we define the semantics of global types using the indistinguishability relation \sim .

► **Definition 2.10** (Semantics of global types). *We construct a state machine $\text{GAut}(\mathbf{G})$ to obtain the semantics of a global type \mathbf{G} . We index every syntactic subterm of \mathbf{G} with a unique index to distinguish common syntactic subterms, denoted with $[G, k]$ for syntactic subterm G and index k . Without loss of generality, the index for \mathbf{G} is 1: $[\mathbf{G}, 1]$. For clarity, we do not quantify indices. We define $\text{GAut}(\mathbf{G}) = (Q_{\text{GAut}(\mathbf{G})}, \Sigma_{\text{sync}}, \delta_{\text{GAut}(\mathbf{G})}, q_{0, \text{GAut}(\mathbf{G})}, F_{\text{GAut}(\mathbf{G})})$ where*

- $Q_{\text{GAut}(\mathbf{G})}$ is the set of all indexed syntactic subterms $[G, k]$ of \mathbf{G}
- $\delta_{\text{GAut}(\mathbf{G})}$ is the smallest set containing $([\sum_{i \in I} p \rightarrow q_i : m_i.[G_i, k_i], k], p \rightarrow q_i : m_i.[G_i, k_i])$ for each $i \in I$, and $([\mu t.[G', k'_2], k'_1], \varepsilon, [G', k'_2])$ and $([t, k'_3], \varepsilon, [\mu t.[G', k'_2], k'_1])$,
- $q_{0, \text{GAut}(\mathbf{G})} = [\mathbf{G}, 1]$, and $F_{\text{GAut}(\mathbf{G})} = \{[0, k] \mid k \text{ is an index for subterm } 0\}$.

We consider asynchronous communication so each interaction is split into its send and receive event. In addition, we consider CSMs as implementation model for global types and, from Lemma 2.9, we know that CSM languages are always closed under the indistinguishability relation \sim . Thus, we also apply its closure to obtain the semantics of \mathbf{G} : $\mathcal{L}(\mathbf{G}) := \mathcal{C}^\sim(\text{split}(\mathcal{L}(\text{GAut}(\mathbf{G}))))$.

The closure $\mathcal{C}^\sim(-)$ corresponds to similar reordering rules in standard MST developments, e.g., [38, Def. 3.2 and 5.3].

► **Example 2.11.** Figure 1a (p.2) illustrates the FSM $\text{GAut}(\mathbf{G}_{2\text{BP}})$. In the following global type, p sends a list of book titles to q : $\mu t.(p \rightarrow q: \text{title}.t + p \rightarrow q: \text{done}.0)$. Its semantics is the union of two cases: if the list of book titles is infinite, i.e., $\mathcal{C}^\sim((p \triangleright q! \text{title}.q \triangleleft p? \text{title})^\omega)$; and the one if the list is finite, i.e., $\mathcal{C}^\sim((p \triangleright q! \text{title}.q \triangleleft p? \text{title})^*. p \triangleright q! \text{done}.q \triangleleft p? \text{done})$. Here, there are only two roles so $\mathcal{C}^\sim(-)$ can solely delay receive events (Rule 4 of \sim).

We distinguish states depending on which subterm they correspond to: *binder states* with their dashed line correspond to a recursion variable binder, while *recursion states* with their dash-dotted lines indicate the use of a recursion variable. We omit ε for transitions from recursion to binder states.

Local Types. For the semantics of local types, we analogously construct a state machine $\text{LAut}(-)$. In contrast, we omit the closure $\mathcal{C}^\sim(-)$ because languages of roles are closed under \sim (cf. [45, Lm. 22]). For the full definition, we refer to the technical report [56]. Compared to global types, we distinguish two more kinds of states for local types: a *send state* (internal choice) has a diamond shape while a *receive state* (external choice) has a rectangular shape. For states with ε as next action, we keep the circular shape and call them *neutral states*. Because of the ε -transitions, Figure 1b (p.2) does not represent the state machine for any local type but illustrates the use of different styles for different kinds of states.

The Implementability Problem for Global Types

The implementability problem for global types asks whether a global type can be implemented in a distributed setting. The projection operator takes the intermediate representation of local types as local specifications for roles. We define implementability directly on the implementation model of CSMs. Intuitively, every collection of local types constitutes a CSM through their semantics.

► **Definition 2.12** (Implementability [45]). *A global type \mathbf{G} is said to be implementable if there exists a deadlock-free CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ such that their languages are the same (protocol fidelity), i.e., $\mathcal{L}(\mathbf{G}) = \mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\})$. We say that $\{\{A_p\}_{p \in \mathcal{P}}\}$ implements \mathbf{G} .*

3 Projection – From Global to Local Types

In this section, we define and visually explain a typical approach to the implementability problem: the *classical projection operator*. It tries to translate global types to local types and, while doing so, checks if this is safe. Behind the scenes, these checks are conducted by a partial merge operator. We consider different variants of the merge operator from the literature and exemplify the features they support. We provide visual explanations of the classical projection operator with these merge operators on the state machines of global types by example. In the technical report [56], we give general descriptions but they are not essential for our observations. Lastly, we summarise the shortcomings of the full merge operator and exemplify them with variants of the two buyer protocol from the introduction.

Classical Projection Operator with Parametric Merge

► **Definition 3.1** (Projection operator). *For a merge operator \sqcap , the projection of a global type \mathbf{G} onto a role $\mathbf{r} \in \mathcal{P}$ is a local type that is defined as follows:² $0 \uparrow_{\mathbf{r}}^\sqcap := 0$ $t \uparrow_{\mathbf{r}}^\sqcap := t$*

$$\left(\sum_{i \in I} \mathbf{p} \rightarrow \mathbf{q}_i : m_i . G_i\right) \uparrow_{\mathbf{r}}^\sqcap := \begin{cases} \oplus_{i \in I} \mathbf{q}_i ! m_i . (G_i \uparrow_{\mathbf{r}}^\sqcap) & \text{if } \mathbf{r} = \mathbf{p} \\ \&_{i \in I} \mathbf{p} ? m_i . (G_i \uparrow_{\mathbf{r}}^\sqcap) & \text{if } \mathbf{r} = \mathbf{q} \\ \sqcap_{i \in I} G_i \uparrow_{\mathbf{r}}^\sqcap & \text{otherwise} \end{cases} \quad (\mu t . G) \uparrow_{\mathbf{r}}^\sqcap := \begin{cases} \mu t . (G \uparrow_{\mathbf{r}}^\sqcap) & \text{if } G \uparrow_{\mathbf{r}}^\sqcap \neq t \\ 0 & \text{otherwise} \end{cases}$$

² The case split for the recursion binder changes slightly across different definitions. We chose a simple but also the least restrictive condition. We simply check whether the recursion is vacuous, i.e. $\mu t . t$, and omit it in this case. We also require to omit μt if t is never used in the result.

Intuitively, a projection operator takes the state machine $\text{GAut}(\mathbf{G})$ for a global type \mathbf{G} and projects each transition label to the respective alphabet of the role, e.g., $p \rightarrow q : m$ becomes $q \triangleleft p ? m$ for role q . This can introduce non-determinism that ought to be resolved by a partial merge operator. Several merge operators have been proposed in the literature.

► **Definition 3.2** (Merge operators). *Let L_1 and L_2 be local types for a role r , and \sqcap be a merge operator. We define different cases for the result of $L_1 \sqcap L_2$:*

$$(1) L_1 \text{ if } L_1 = L_2 \quad (2) \left(\begin{array}{cc} \&_{i \in I \setminus J} q ? m_i . L'_{1,i} & \& \\ \&_{i \in I \cap J} q ? m_i . (L'_{1,i} \sqcap L'_{2,i}) & \& \\ \&_{i \in J \setminus I} q ? m_i . L'_{2,i} & \end{array} \right) \text{ if } \begin{cases} L_1 = \&_{i \in I} q ? m_i . L'_{1,i}, \\ L_2 = \&_{i \in J} q ? m_i . L'_{2,i} \end{cases}$$

$$(3) \mu t_1 . (L'_1 \sqcap L'_2 [t_2/t_1]) \text{ if } L_1 = \mu t_1 . L'_1 \text{ and } L_2 = \mu t_2 . L'_2$$

Each merge operator is defined by a collection of cases it can apply. If none of the respective cases applies, the result of the merge is undefined. The plain merge \sqcap [23] can only apply Case (1). The semi-full merge \sqcap [64] can apply Cases (1) and (2). The full merge \sqcap [54] can apply all Cases (1), (2), and (3).

We will also consider the availability merge operator \sqcap by Majumdar et al. [45] which builds on the full merge operator but generalises Case (2) to allow sender-driven choice. We will explain the main differences in Remark 3.12.

► **Remark 3.3** (Correctness of projection). This would be the correctness criterion for projection: Let \mathbf{G} be some global type and let plain merge \sqcap , semi full merge \sqcap , full merge \sqcap , or availability merge \sqcap be the merge operator \sqcap . If $\mathbf{G} \uparrow_p^\sqcap$ is defined for each role p , then the CSM $\{\{\text{LAut}(\mathbf{G} \uparrow_p^\sqcap)\}\}_{p \in \mathcal{P}}$ implements \mathbf{G} .

We do not actually prove this so we do not state it as lemma. *But why does this hold?*

The implementability condition is the combination of deadlock freedom and protocol fidelity. Coppo et al. [23] show that *subject reduction* entails protocol fidelity and progress while progress, in turn, entails deadlock freedom. Subject reduction has been proven for the plain merge operator [23, Thm. 1] and the semi-full operator [64, Thm. 1]. Scalas and Yoshida pointed out that several versions of classical projection with the full merge are flawed [54, Sec. 8.1]. Hence, we have chosen a full merge operator whose correctness follows from the correctness of the more general availability merge operator. For the latter, correctness was proven by Majumdar et al. [45, Thm. 16].

► **Example 3.4** (Projection without merge / Collapsing erasure). In the introduction, we considered $\mathbf{G}_{2\text{BP}}$ and the FSM for its semantics in Figure 1a. We projected (without merge) onto seller s to obtain the FSM in Figure 1b. In general, we also collapse neutral states with a single ε -transition and their only successor. We call this *collapsing erasure*. We only need to actually collapse states for the protocol in Figure 4a. In all other illustrations, we indicate the interactions the role is not involved with the following notation: $[p \rightarrow q : l] \rightsquigarrow \varepsilon$.

On the Structure of $\text{GAut}(\mathbf{G})$

We now show that the state machine for every local and global type has a certain shape. This simplifies the visual explanations of the different merge operators. Intuitively, every such state machine has a tree-like structure where backward transitions only happen at leaves of the tree, are always labelled with ε , and only lead to ancestors. The FSM in Figure 1a (p.2) illustrates this shape where the root of the tree is at the top.

► **Definition 3.5** (Ancestor-recursive, non-merging, intermediate recursion, etc.). *Let $A = (Q, \Delta, \delta, q_0, F)$ be a finite state machine. We say that A is ancestor-recursive if there is a function $\text{lvl}: Q \rightarrow \mathbb{N}$ such that, for every transition $q \xrightarrow{x} q' \in \delta$, one of the two holds:*

- (a) $\text{lvl}(q) > \text{lvl}(q')$, or
 (b) $x = \varepsilon$ and there is a run from the initial state q_0 (without going through q) to q' which can be completed to reach q : $q_0 \xrightarrow{\bar{}} \dots \xrightarrow{\bar{}} q_n$ is a run with $q_n = q'$ and $q \neq q_i$ for every $0 \leq i \leq n$, and the run can be extended to $q_0 \xrightarrow{\bar{}} \dots \xrightarrow{\bar{}} q_n \xrightarrow{\bar{}} \dots \xrightarrow{\bar{}} q_{n+m}$ with $q_{n+m} = q$. Then, the state q' is called ancestor of q .

We call the first (a) kind of transition forward transition while the second (b) kind is a backward transition. The state machine A is said to be free from intermediate recursion if every state q with more than one outgoing transition, i.e., $|\{q' \mid q \xrightarrow{\bar{}} q' \in \delta\}| > 1$, has only forward transitions. We say that A is non-merging if every state only has one incoming edge with greater level, i.e., for every state q' , $\{q \mid q \xrightarrow{\bar{}} q' \in \delta \wedge \text{lvl}(q) > \text{lvl}(q')\} \leq 1$. The state machine A is dense if, for every $q \xrightarrow{x} q' \in \delta$, the transition label x is ε implies that q has only one outgoing transition. Last, the cone of q are all states q' which are reachable from q and have a smaller level than q , i.e., $\text{lvl}(q) > \text{lvl}(q')$.

► **Proposition 3.6** (Shape of $\text{GAut}(\mathbf{G})$ and $\text{LAut}(L)$). *Let \mathbf{G} be some global type and L be some local type. Then, both $\text{GAut}(\mathbf{G})$ and $\text{LAut}(L)$ are ancestor-recursive, free from intermediate recursion, non-merging, and dense.*

For both, the only forward ε -transitions occur precisely from binder states while backward transitions happen from variable states to binder states. The illustrations for our examples always have the initial state, which is the state with the greatest level, at the top. This is why we use greater and higher as well as smaller and lower interchangeably for levels.

Features of Different Merge Operators by Example

In this section, we exemplify which features each of the merge operators supports. We present a sequence of implementable global types. Despite, some cannot be handled by some (or all) merge operators. If a global type is not projectable using some merge operator, we say it is *rejected* and it constitutes a *negative* example for this merge operator. We focus on role r when projecting. Thus, rejected mostly means that there is (at least) no projection onto r . If a global type is projectable by some merge operator, we call it a *positive* example. All examples strive for minimality and follow the idea that roles decide whether to take a left (l) or right (r) branch of a choice.

► **Example 3.7** (Positive example for plain merge). The following global type is implementable:

$$\mu t. + \left\{ \begin{array}{l} p \rightarrow q : l. (q \rightarrow r : l. 0 + q \rightarrow r : r. t) \\ p \rightarrow q : r. (q \rightarrow r : l. 0 + q \rightarrow r : r. t) \end{array} \right.$$

The state machine for its semantics is given in Figure 2a. After collapsing erasure, there is a non-deterministic choice from q'_0 leading to q_1 and q_4 since r is not involved in the initial choice. The plain merge operator can resolve this non-determinism since both cones of q_1 and q_4 represent the same subterm. Technically, there is an isomorphism between the states in both cones which preserves the kind of states as well as the transition labels and the backward transitions from isomorphic recursion states lead to the same binder state. The result is illustrated in Figure 2b. It is also the FSM of a local type for r which is the result of the (syntactic) plain merge: $\mu t. (q?l.0 \ \& \ q?r.t)$.

Our explanation on FSMs allows to check congruence of cones to merge while the definition requires syntactic equality. If we swap the order of branches $q \rightarrow r : l$ and $q \rightarrow r : r$ in Figure 2a on the right, the syntactic merge rejects. Still, because both are semantically the same protocol specification, we expect tools to check for such easy fixes.

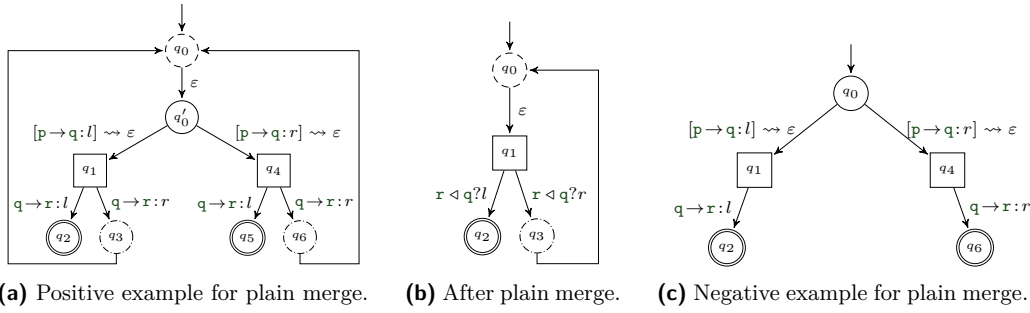


Figure 2 The FSM on the left represents an implementable global type that is accepted by plain merge. It implicitly shows the FSM after collapsing erasure: every interaction r is not involved in is given as $[p \rightarrow q:l] \rightsquigarrow \epsilon$. The FSM in the middle is the result of the plain merge. The FSM on the right represents an implementable global type that is rejected by plain merge. It is obtained from the left one by removing one choice option in each branch of the initial choice.

► **Example 3.8** (Negative example for plain merge). We consider the following simple implementable global type where the choice by p is propagated by q to r : $\mu t. + \left\{ \begin{array}{l} p \rightarrow q:l. q \rightarrow r:l. 0 \\ p \rightarrow q:r. q \rightarrow r:r. 0 \end{array} \right.$. The corresponding state machine is illustrated in Figure 2c. Here, q_0 exhibits non-determinism but the plain merge fails because q_1 and q_4 have different outgoing transition labels.

Intuitively, the plain merge operator forbids that any, but the two roles involved in a choice, can have different behaviour after the choice. It basically forbids propagating a choice. The semi-full merge overcomes this shortcoming and can handle the previous example. We present a slightly more complex one to showcase the features it supports.

► **Example 3.9** (Positive example for semi-full merge). Let us consider this implementable global type: $\mu t. + \left\{ \begin{array}{l} p \rightarrow q:l. (q \rightarrow r:l. 0 + q \rightarrow r:m. 0) \\ p \rightarrow q:r. (q \rightarrow r:m. 0 + q \rightarrow r:r. t) \end{array} \right.$, illustrated in Figure 3a. After applying collapsing erasure, there is a non-deterministic choice from q_0 leading to q_1 and q_4 since r is not involved in the initial choice, We apply the semi-full merge for both states. Both are receive states so Case (2) applies. First, we observe that $r \triangleleft q?l$ and $r \triangleleft q?r$ are unique to one of the two states so both transitions, with the cones of the states they lead to, can be kept. Second, there is $r \triangleleft q?m$ which is common to both states. We recursively apply the semi-full merge and, with Case (1), observe that the result $q_{3|5}$ is simply a final state. Overall, we obtain the state machine in Figure 3b, which is equivalent to the result of the syntactic projection with semi-full merge: $\mu t. (q?l. 0 \ \& \ q?m. 0 \ \& \ q?r. t)$.

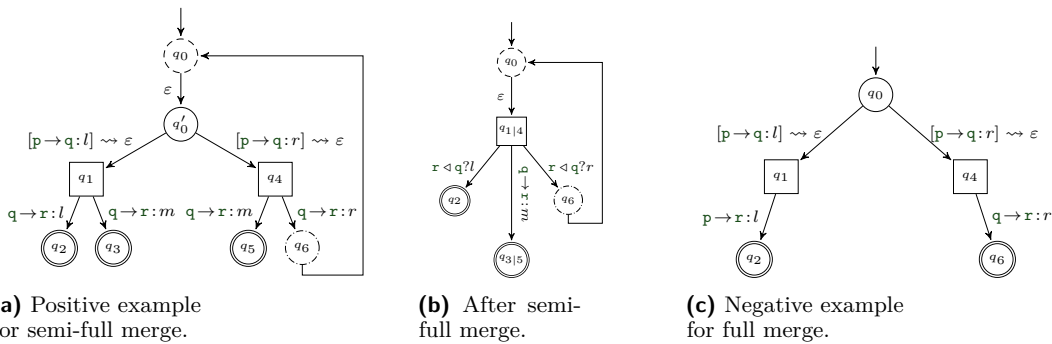
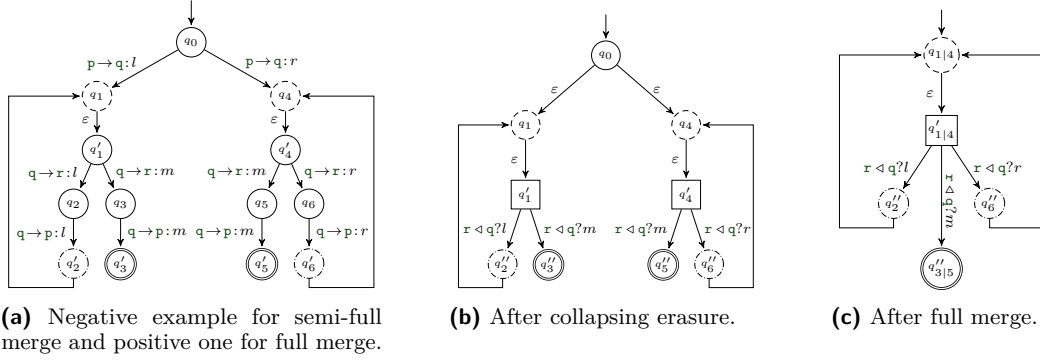


Figure 3 The FSM on the left represents an implementable global type (and implicitly the collapsing erasure onto r) that is accepted by semi-full merge. The FSM in the middle is the result of the semi-full merge. The FSM on the right is a negative example for the full merge operator.



■ **Figure 4** The FSM on the left represents an implementable global type that is rejected by the semi-full merge. It is accepted by the full merge: collapsing erasure yields the FSM in the middle and applying the full merge the FSM on the right.

► **Example 3.10** (Negative example for semi-full merge and positive example for full merge). The semi-full merge operator rejects the following implementable global type:

$$+ \left\{ \begin{array}{l} p \rightarrow q:l. \mu t_1. (q \rightarrow r:l. q \rightarrow p:l. t_1 + q \rightarrow r:m. q \rightarrow p:m. 0) \\ p \rightarrow q:r. \mu t_2. (q \rightarrow r:m. q \rightarrow p:m. 0 + q \rightarrow r:r. q \rightarrow p:r. t_2) \end{array} \right.$$

Its FSM and the FSM after collapsing erasure is given in Figures 4a and 4b. Intuitively, it would need to recursively merge the parts after both recursion binders in order to merge the branches with receive event $r \triangleleft q?m$ but it cannot do so. The full merge can handle this global type. It can descend beyond q_1 and q_4 and is able to merge q'_1 and q'_4 . To obtain $q''_{3|5}$, it applies Case (1) while $q'_{1|4}$ is only feasible with Case (2). The result is embedded into the recursive structure to obtain the FSM in Figure 4c. It is equivalent to the (syntactic) result, which renames the recursion variable for one branch: $\mu t_1. (q?l. t_1 \& q?m. 0 \& q?r. t_1)$.

► **Example 3.11** (Negative example for full merge). We consider a simple implementable global type where p propagates its decision to r in the top branch while q propagates it in the bottom branch: $+ \left\{ \begin{array}{l} p \rightarrow q:l. p \rightarrow r:l. 0 \\ p \rightarrow q:r. q \rightarrow r:r. 0 \end{array} \right.$. It is illustrated in Figure 3c. This cannot be projected onto r by the full merge operator for which all receive events need to have the same sender.

► **Remark 3.12** (On sender-driven choice). Majumdar et al. [45] proposed a classical projection operator that overcomes this shortcoming. It can project the previous example. In general, allowing to receive from different senders has subtle consequences. Intuitively, messages from different senders could overtake each other in a distributed setting and one cannot rely on the FIFO order provided by the channel of a single sender. Majumdar et al. employ a message availability analysis to ensure that there cannot be any confusion about which branch shall be taken. Except for the possibility to merge cases where a receiver receives from multiple senders, their merge operator suffers from the same shortcomings as all classical projection operators. For details, we refer to their work [45].

Shortcomings of Classical Projection/Merge Operators

We present slight variations of the two buyer protocol that are implementable but rejected by all of the presented projection/merge operators.

► **Example 3.13.** We obtain an implementable variant by omitting both message interactions $a \rightarrow s: no$ with which buyer a notifies seller s that they will not buy the item:

$$\mu t. + \left\{ \begin{array}{l} a \rightarrow s: query. s \rightarrow a: price. (a \rightarrow b: split. (b \rightarrow a: yes. a \rightarrow s: buy. t + b \rightarrow a: no. t) + a \rightarrow b: cancel. t) \\ a \rightarrow s: done. a \rightarrow b: done. 0 \end{array} \right.$$

This global type cannot be projected onto seller s . The merge operator would need to merge a recursion variable with an external choice. Visually, the merge operator does not allow to unfold the variable t and try to merge again. However, there is a local type for seller s :

$$\mu t_1. \& \begin{cases} a?query. \mu t_2. a!price. (a?buy. t_1 \& a?query. t_2 \& a?done. 0) \\ a?done. 0 \end{cases}$$

The local type has two recursion variable binders while the global type only has one. Classical projection operators can never yield such a structural change: the merge operator can only merge states but not introduce new ones or introduce new backward transitions.

► **Example 3.14** (Two Buyer Protocol with Subscription). In this variant, buyer a first decides whether to subscribe to a yearly discount offer or not – before purchasing the sequence of items – and notifies buyer b if it does so: $\mathbf{G}_{2BPWS} := + \begin{cases} a \rightarrow s: login. \mathbf{G}_{2BP} \\ a \rightarrow s: subscribe. a \rightarrow b: subscribed. \mathbf{G}_{2BP} \end{cases}$. The merge operator needs to merge a recursion variable binder μt with the external choice $b \triangleleft a?subscribed$. Still, there is a local type L_b for b such that $\mathcal{L}(L_b) = \mathcal{L}(\mathbf{G}_{2BPWS}) \downarrow_{\Sigma_b}$:

$$L_b := \& \begin{cases} a?split. (a!yes. L(t_1) \oplus a!no. L(t_2)) \\ a?cancel. L(t_3) \\ a?done. 0 \\ a?subscribed. L(t_4) \end{cases} \quad \text{where } L(t) := \mu t. \& \begin{cases} a?split. (a!yes. t \oplus a!no. t) \\ a?cancel. t \\ a?done. 0 \end{cases}$$

In fact, one can also rely on the fact that buyer a will comply with the intended protocol. Then, it suffices to introduce one recursion variable t in the beginning and substitute every $L(-)$ with t , yielding a local type L'_b with $\mathcal{L}(L_b) \subseteq \mathcal{L}(L'_b)$.

Similarly, classical projection operator cannot handle global types where choices can be disambiguated with semantic properties, e.g., counting modulo a constant. Scalas and Yoshida [54] also identified another shortcoming: most classical projection operators require all branches of a loop to contain the same set of active roles. Thus, they cannot project the following global type. It is implementable and if it was projectable, the result would be equivalent to the local types given in their example [54, Fig. 4 (2)].

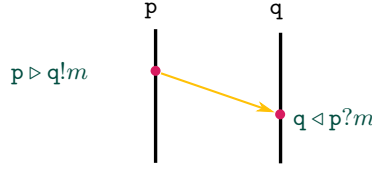
► **Example 3.15** (Two Buyer Protocol with Inner Recursion). This variant allows to recursively negotiate how to split the price (and omits the outer recursion):

$$\mathbf{G}_{2BPIR} := a \rightarrow s: query. s \rightarrow a: price. \mu t. + \begin{cases} a \rightarrow b: split. (b \rightarrow a: yes. a \rightarrow s: buy. 0 + b \rightarrow a: no. t) \\ a \rightarrow b: cancel. a \rightarrow s: no. 0 \end{cases}$$

These shortcomings have been addressed by some non-classical approaches. For example, Scalas and Yoshida [54] employ model checking while Castagna et al. [16] characterise implementable global types with an undecidable well-formedness condition and give a sound algorithmically checkable approximation. It is not known whether the implementability problem for global types, neither with directed or sender-driven choice, is decidable. We answer this question positively for the more general case of sender-driven choice.

4 Implementability for Global Types from MSTs is Decidable

In this section, we show decidability of the implementability problem for global types with sender-driven choice, using results from the domain of message sequence charts. We introduce high-level message sequence charts (HMSCs) and recall an HMSC encoding for global types. In general, implementability for HMSCs is undecidable but we show that global types, when encoded as HMSCs, belong to a class of HMSCs for which implementability is decidable.



■ **Figure 5** Highlighting the elements of an MSC $(N, p, f, l, (\leq_p)_{p \in \mathcal{P}})$ [57, Fig. 3].

4.1 High-level Message Sequence Charts

Our definitions of (high-level) message sequence charts follow work by Genest et al. [30] and Stutz and Zufferey [57]. If reasonable, we adapt terminology to the MST setting.

► **Definition 4.1** (Message Sequence Charts). A message sequence chart (MSC) is a 5-tuple $M = (N, p, f, l, (\leq_p)_{p \in \mathcal{P}})$ where

- N is a set of send (S) and receive (R) event nodes such that $N = S \uplus R$ (where \uplus denotes disjoint union),
- $p: N \rightarrow \mathcal{P}$ maps each event node to the role acting on it,
- $f: S \rightarrow R$ is an injective function linking corresponding send and receive event nodes,
- $l: N \rightarrow \Sigma$ labels every event node with an event, and
- $(\leq_p)_{p \in \mathcal{P}}$ is a family of total orders for the event nodes of each role: $\leq_p \subseteq p^{-1}(p) \times p^{-1}(p)$.

We visually highlight the elements of an MSC in Figure 5.

An MSC M induces a partial order \leq_M on N that is defined co-inductively:

$$\frac{e \leq_p e'}{e \leq_M e'} \text{ PROC} \quad \frac{s \in S}{s \leq_M f(s)} \text{ SND-RCV} \quad \frac{}{e \leq_M e} \text{ REFL} \quad \frac{e \leq_M e' \quad e' \leq_M e''}{e \leq_M e''} \text{ TRANS}$$

The labelling function l respects the function f : for every send event node e , we have that $l(e) = p(e) \triangleright p(f(e))!m$ and $l(f(e)) = p(f(e)) \triangleleft p(e)?m$ for some $m \in \mathcal{V}$.

All MSCs in our work respect FIFO, i.e., there are no p and q such that there are $e_1, e_2 \in p^{-1}(p)$ with $e_1 \neq e_2$, $l(e_1) = l(e_2)$, $e_1 \leq_p e_2$ and $f(e_2) \leq_q f(e_1)$ (also called degenerate) and for every pair of roles p, q , and for every two event nodes $e_1 \leq_M e_2$ with $l(e_i) = p \triangleright q! _$ for $i \in \{1, 2\}$, it holds that $\mathcal{V}(w_p) = \mathcal{V}(f(w_p))$ where w_p is the (unique) linearisation of $p^{-1}(p)$. A *basic MSC* (BMSC) has a finite number of nodes N and \mathcal{M} denotes the set of all BMSCs. When unambiguous, we omit the index M for \leq_M and write \leq . We define \leq as expected. The language $\mathcal{L}(M)$ of an MSC M collects all words $l(w)$ for which w is a linearisation of N that is compliant with \leq_M .

If one thinks of a BMSC as straight-line code, a high-level message sequence chart adds control flow. It embeds BMSCs into a graph structure which allows for choice and recursion.

► **Definition 4.2** (High-level Message Sequence Charts). A high-level message sequence chart (HMSC) is a 5-tuple (V, E, v^I, V^T, μ) where V is a finite set of vertices, $E \subseteq V \times V$ is a set of directed edges, $v^I \in V$ is an initial vertex, $V^T \subseteq V$ is a set of terminal vertices, and $\mu: V \rightarrow \mathcal{M}$ is a function mapping every vertex to a BMSC. A path in an HMSC is a sequence of vertices v_1, \dots from V that is connected by edges, i.e., $(v_i, v_{i+1}) \in E$ for every i . A path is maximal if it is infinite or ends in a vertex from V^T .

Intuitively, the language of an HMSC is the union of all languages of the finite and infinite MSCs generated from maximal paths in the HMSC and is formally defined in the technical report [56]. Like global types, an HMSC specifies a protocol. The implementability question was also posed for HMSCs and studied as *safe realisability*. If the CSM is not required to be deadlock-free, it is called weak realisability.

► **Definition 4.3** (Safe realisability of HMSCs [3]). *An HMSC H is said to be safely realisable if there exists a deadlock-free CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ such that $\mathcal{L}(H) = \mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\})$.*

Encoding Global Types from MSTs as HMSCs

Stutz and Zufferey [57, Sec. 5.2] provide a formal encoding $H(-)$ from global types to HMSCs. We refer to the technical report [56] for the definition. We adapt their correctness result to our setting. In particular, our semantics of \mathbf{G} use the closure operator $\mathcal{C}^\sim(-)$ while they distinguish between a type and execution language. We also omit the closure operator on the right-hand side because HMSCs are closed with regard to this operator [57, Lm. 5].

► **Theorem 4.4.** *Let \mathbf{G} be a global type. Then, the following holds: $\mathcal{L}(\mathbf{G}) = \mathcal{L}(H(\mathbf{G}))$.*

4.2 Implementability is Decidable

We introduce a mild assumption for global types. Intuitively, we require that every run of the protocol can always terminate but does not need to. Basically, this solely rules out global types that have loops without exit (cf. Example 4.20). In practice, it is reasonable to assume a mechanism to terminate a protocol for maintenance for instance. Note that this assumption constitutes a structural property of a protocol and no fairness condition on runs of the protocol.

► **Assumption** (0-Reachable). *We say a global type \mathbf{G} is 0-reachable if every prefix of a word in its language can be completed to a finite word in its language. Equivalently, we require that the vertex for the syntactic subterm 0 is reachable from any vertex in $H(\mathbf{G})$.*

The MSC approach to safe realisability for HMSCs is different from the classical projection approach to implementability. Given an HMSC, there is a canonical candidate implementation which always implements the HMSC if an implementation exists [2, Thm. 13]. Therefore, approaches center around checking safe realisability of HMSC languages and establishing conditions on HMSCs that entail safe realisability.

► **Definition 4.5** (Canonical candidate implementation [2]). *Given an HMSC H and a role p , let $A'_p = (Q', \Sigma_p, \delta', q'_0, F')$ be a state machine with $Q' := \{q_w \mid w \in \text{pref}(\mathcal{L}(H) \downarrow_{\Sigma_p})\}$, $F' := \{q_w \mid w \in \mathcal{L}_{\text{fin}}(H) \downarrow_{\Sigma_p}\}$, and $\delta'(q_w, x, q_{wx})$ for $x \in \Sigma_{\text{async}}$. The resulting state machine A'_p is not necessarily finite so A'_p is determinised and minimised which yields the FSM A_p . We call $\{\{A_p\}_{p \in \mathcal{P}}\}$ the canonical candidate implementation of H .*

Intuitively, the intermediate state machine A'_p constitutes a tree whose maximal finite paths give $\mathcal{L}(H) \downarrow_{\Sigma_p} \cap \Sigma_p^*$. This set can be infinite and, thus, the construction might not be effective. We give an effective construction of a deterministic FSM for the same language which was very briefly hinted at by Alur et al. [3, Proof of Thm. 3].

► **Definition 4.6** (Projection by Erasure). *Let p be a role and $M = (N, p, f, l, (\leq_p)_{p \in \mathcal{P}})$ be an MSC. We denote the set of nodes of p with $N_p := \{n \mid p(n) = p\}$ and define a two-ary next-relation on N_p : $\text{next}(n_1, n_2)$ iff $n_1 \leq_p n_2$ and there is no n' with $n_1 \leq_p n' \leq_p n_2$. We define the projection by erasure of M on to p : $M \downarrow_p = (Q_M, \Sigma_p, \delta_M, q_{M,0}, \{q_{M,f}\})$ with*

$$Q_M := \{q_n \mid n \in N_p\} \uplus \{q_{M,0}\} \uplus \{q_{M,f}\} \text{ and}$$

$$\delta_M := \{q_{M,0} \xrightarrow{\varepsilon} q_{n_1} \mid \forall n_2. n_1 \leq n_2\} \uplus \{q_{n_1} \xrightarrow{l(n_1)} q_{n_2} \mid \text{next}(n_1, n_2)\} \uplus \{q_{n_2} \xrightarrow{l(n_2)} q_{M,f} \mid \forall n_1. n_1 \leq n_2\}$$

where \uplus denotes disjoint union. Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. We construct the projection by erasure for every vertex and identify them with the vertex, e.g., Q_v instead of $Q_{\mu(v)}$. We construct an auxiliary FSM $(Q'_H, \Sigma_p, \delta'_H, q'_{H,0}, F'_H)$ with $Q'_H = \biguplus_{v \in V} Q_v$, $\delta'_H = \biguplus_{v \in V} \delta_v \uplus \{q_{v_1, f} \xrightarrow{\varepsilon} q_{v_2, 0} \mid (v_1, v_2) \in E\}$, $q'_{H,0} = q_{v^I, 0}$, and $F'_H = \biguplus_{v \in V^F} q_{v, f}$. We determinise and minimise $(Q'_H, \Sigma_p, \delta'_H, q'_{H,0}, F'_H)$ to obtain $H \downarrow_p := (Q_H, \Sigma_p, \delta_H, q_{H,0}, F_H)$, which we define to be the projection by erasure of H onto p . The CSM formed from the projections by erasure $\{\{H \downarrow_p\}\}_{p \in \mathcal{P}}$ is called erasure candidate implementation.

► **Lemma 4.7** (Correctness of Projection by Erasure). *Let H be an HMSC, p be a role, and $H \downarrow_p$ be its projection by erasure. Then, the following language equality holds: $\mathcal{L}(H \downarrow_p) = \mathcal{L}(H) \downarrow_{\Sigma_p}$.*

The proof is straightforward and can be found in the technical report [56]. From this result and the construction of the canonical candidate implementation, it follows that the projection by erasure admits the same finite language.

► **Corollary 4.8.** *Let H be an HMSC, p be a role, $H \downarrow_p$ be its projection by erasure, and A_p be the canonical candidate implementation. Then, it holds that $\mathcal{L}_{\text{fin}}(H \downarrow_p) = \mathcal{L}_{\text{fin}}(A_p)$.*

The projection by erasure can be computed effectively and is deterministic. Thus, we use it in place of the canonical candidate implementation. Given a global type, the erasure candidate implementation for its HMSC encoding implements it if it is implementable.

► **Theorem 4.9.** *Let \mathbf{G} be a global type and $\{\{H(\mathbf{G}) \downarrow_p\}\}_{p \in \mathcal{P}}$ be its erasure candidate implementation. If $\mathcal{L}_{\text{fin}}(\mathbf{G})$ is implementable³, then $\{\{H(\mathbf{G}) \downarrow_p\}\}_{p \in \mathcal{P}}$ is deadlock-free and $\mathcal{L}_{\text{fin}}(\{\{H(\mathbf{G}) \downarrow_p\}\}_{p \in \mathcal{P}}) = \mathcal{L}_{\text{fin}}(\mathbf{G})$.*

This result does only account for finite languages so we extend it for infinite sequences. For both, the proof can be found in the technical report [56].

► **Lemma 4.10** ("Finite implementation" generalises to infinite language for 0-reachable global types). *Let \mathbf{G} be a 0-reachable global type and $\{\{A_p\}\}_{p \in \mathcal{P}}$ be an implementation for $\mathcal{L}_{\text{fin}}(\mathbf{G})$. Then, it holds that $\mathcal{L}_{\text{inf}}(\{\{A_p\}\}_{p \in \mathcal{P}}) = \mathcal{L}_{\text{inf}}(\mathbf{G})$, and, thus, $\mathcal{L}(\{\{A_p\}\}_{p \in \mathcal{P}}) = \mathcal{L}(\mathbf{G})$.*

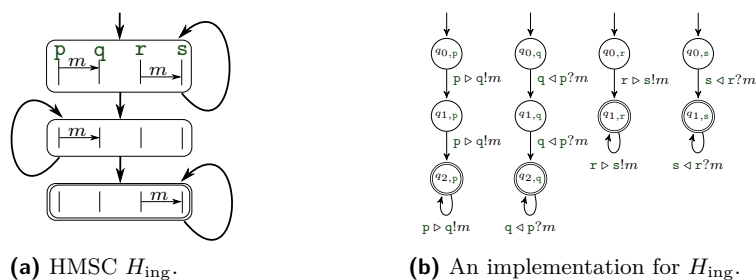
► **Corollary 4.11.** *Let \mathbf{G} be a 0-reachable implementable global type. Then, the erasure candidate implementation $\{\{H(\mathbf{G}) \downarrow_p\}\}_{p \in \mathcal{P}}$ implements \mathbf{G} .*

So far, we have shown that, if \mathbf{G} is implementable, the erasure candidate implementation for its HMSC encoding $H(\mathbf{G})$ implements \mathbf{G} . For HMSCs, this is undecidable in general [44]. We show that, because of their syntactic restrictions on choice, global types fall into the class of globally-cooperative HMSCs for which implementability is decidable.

► **Definition 4.12** (Communication graph [31]). *Let $M = (N, p, f, l, (\leq_p)_{p \in \mathcal{P}})$ be an MSC. The communication graph of M is a directed graph with node p for every role p that sends or receives a message in M and edges $p \rightarrow q$ if M contains a message from p to q , i.e., there is $e \in N$ such that $p(e) = p$ and $p(f(e)) = q$.*

It is important that the communication graph of M does not have a node for every role but only the active ones, i.e., the ones that send or receive in M .

³ Implementability is lifted to languages as expected.



■ **Figure 6** An implementable HMSC which is not globally-cooperative with its implementation.

► **Definition 4.13** (Globally-cooperative HMSCs [31]). *An HMSC $H = (V, E, v^I, V^T, \mu)$ is called globally-cooperative if for every loop, i.e., v_1, \dots, v_n with $(v_i, v_{i+1}) \in E$ for every $1 \leq i < n$ and $(v_n, v_1) \in E$, the communication graph of $\mu(v_1) \dots \mu(v_n)$ is weakly connected, i.e., all nodes are connected if every edge is considered undirected.*

We can check this directly for a global type \mathbf{G} . It is straightforward to define a communication graph for words from Σ_{sync}^* . We check it on $\text{GAut}(\mathbf{G})$: for each binder state, we check the communication graph for the shortest trace to every corresponding recursion state.

► **Theorem 4.14** (Thm. 3.7 [44]). *Let H be a globally-cooperative HMSC. Restricted to its finite language $\mathcal{L}_{\text{fin}}(H)$, safe realisability is EXPSPACE-complete.*

► **Lemma 4.15.** *Let \mathbf{G} be an implementable 0-reachable global type. Then, its HMSC encoding $H(\mathbf{G})$ is globally-cooperative.*

The proof can be found in the technical report [56] and is far from trivial. We explain the main intuition for the proof with the following example where we exemplify why the same result does not hold for HMSCs in general.

► **Example 4.16** (Implementable HMSC but not globally cooperative). HMSC H_{ing} in Figure 6a is implementable but neither globally-cooperative nor representable with a global type. In the first loop, p sends a message m to q while r sends a message m to s so the communication graph is not weakly connected. In the second loop, only the interaction between p and q is specified, while, in the third one, it is only the one between r and s . For a variant of the protocol without the second loop, any candidate implementation can always expose an execution with more interactions between p and q than the ones between r and s , due to the lack of synchronisation. Here, the second loop can make up for such executions so any execution has a path in H_{ing} . The CSM in Figure 6b implements H_{ing} . In the technical report [56], we explain in detail why there is a path in H_{ing} for any trace of the CSM and how to modify the example not to have final states with outgoing transitions.

► **Theorem 4.17.** *Checking implementability of 0-reachable global types with sender-driven choice is in EXPSPACE.*

Proof. Let \mathbf{G} be a 0-reachable global type with sender-driven choice. We construct $H(\mathbf{G})$ from \mathbf{G} and check if it is globally-cooperative. For this, we apply the coNP-algorithm by Genest et al. [31] which is based on guessing a subgraph and checking its communication graph. If $H(\mathbf{G})$ is not globally cooperative, we know from Lemma 4.15 that \mathbf{G} is not implementable. If $H(\mathbf{G})$ is globally cooperative, we check safe realisability for $H(\mathbf{G})$. By Theorem 4.14, this is in EXPSPACE. If $H(\mathbf{G})$ is not safely realisable, it trivially follows that \mathbf{G} is not implementable. If $H(\mathbf{G})$ is safely realisable, \mathbf{G} is implemented by the erasure candidate implementation with Theorem 4.9 and Lemma 4.10. ◀

Thus, the implementability problem for global types with sender-driven choice is decidable.

► **Corollary 4.18.** *Let \mathbf{G} be a 0-reachable global type with sender-driven choice. It is decidable whether \mathbf{G} is implementable and there is an algorithm to obtain its implementation.*

► **Remark 4.19 (Progress).** The property deadlock freedom is sometimes also studied as *progress* – in the sense that a system should never get stuck. For infinite executions, however, a role could starve in a non-final state by waiting for a message that is never sent [16, Sec. 3.2]. Thus, Castagna et al. [16] consider a stronger notion of progress (Def. 3.3: live session) which requires that every role could eventually reach a final state. Our results also apply to this stronger notion of progress, which entails that any sent message can eventually be received. The notion only requires it to be possible but we can ensure that no role starves in a non-final state in two ways. First, we can impose a (strong) fairness assumption – as Castagna et al. [16]. Second, we can require that every loop branch contains at least all roles that occur in interactions of any path with which the protocol can finish.

The Odd Case of Infinite Loops Without Exits. In theory, one can think of protocols for which the 0-Reachability-Assumption (p.15) does not hold. They would simply recurse indefinitely and can never terminate. This allows interesting behaviour like two sets of roles that do not interact with each other as the following example shows.

► **Example 4.20.** Consider the following global type: $\mathbf{G} = \mu t. p \rightarrow q : m. r \rightarrow s : m. t$. This is basically the protocol that consists only of the first loop of H_{ing} (Example 4.16). It describes an infinite execution with two pairs of roles that independently send and receive messages. This can be implemented in an infinite setting but the loop can never be exited due to the lack of synchronisation, breaking protocol fidelity upon termination.

Expressiveness of Local Types. Local types also have a distinct expression for termination: 0. Thus, if one considers the FSM of a local type, every final state has no outgoing transition. Our proposed algorithm might yield FSMs for which this is not the case. However, the language of such an FSM cannot be represented as local type since both our construction and FSMs for local types are deterministic. The latter are also ancestor-recursive, free of intermediate recursion, non-merging and dense (Proposition 3.6). For FSMs from our procedure, this is not the case but the ones without final states with outgoing transitions could possibly be transformed to local types, making subtyping techniques applicable. One could also study subtyping for FSMs as local specifications. We leave both for future work.

On Lower Bounds for Implementability. For general globally-cooperative HMSCs, i.e., that are not necessary the encoding of a global type, safe realisability is EXPSPACE-hard [44]. This hardness result does not carry over for the HMSC encoding $H(\mathbf{G})$ of a global type \mathbf{G} . The construction exploits that HMSCs do not impose any restrictions on choice. Global types, however, require every branch to be chosen by a single sender.

5 MSC Techniques for MST Verification

In the previous section, we generalised results from the MSC literature to show decidability of the implementability problem for global types from MSTs, yielding an EXPSPACE-algorithm. In this section, we consider further restrictions on HMSCs to obtain algorithms with better complexity for global types. First, we transfer the algorithms for \mathcal{I} -closed HMSCs, which requires an HMSC not to exhibit certain anti-patterns of communication, to global types.

Second, we explain approaches for HMSCs that introduced the idea of choice to HMSCs and a characterisation of implementable MSC languages. Third, we present a variant of the implementability problem. It can make unimplementable global types implementable without changing a protocol's structure. From now on, we may use the term implementability for HMSCs instead of safe realisability.

\mathcal{I} -closed Global Types

For globally-cooperative HMSCs, the implementability problem is EXPSPACE-complete. The membership in EXPSPACE was shown by reducing the problem to implementability of \mathcal{I} -closed HMSCs [44, Thm. 3.7]. These require the language of an HMSC to be closed with regard to an independence relation \mathcal{I} , where, intuitively, two interactions are independent if there is no role which is involved in both. Implementability for \mathcal{I} -closed HMSCs is PSPACE-complete [44, Thm. 3.6]. As for the EXPSPACE-hardness for globally-cooperative HMSCs, the PSPACE-hardness exploits features that cannot be modelled with global types and there might be algorithms with better worst-case complexity.

We adapt the definitions [44] to the MST setting. These consider atomic BMSCs, which are BMSCs that cannot be split further. With the HMSC encoding for global types, it is straightforward that atomic BMSCs correspond to individual interactions for global types. Thus, we define the independence relation \mathcal{I} on the alphabet Σ_{sync} .

► **Definition 5.1** (Independence relation \mathcal{I}). *We define the independence relation \mathcal{I} on Σ_{sync} :*

$$\mathcal{I} := \{(p \rightarrow q : m, r \rightarrow s : m') \mid \{p, q\} \cap \{r, s\} = \emptyset\}.$$

We lift this to an equivalence relation $\equiv_{\mathcal{I}}$ on words as its transitive and reflexive closure:

$$\equiv_{\mathcal{I}} := \{(u.x_1.x_2.w, u.x_2.x_1.w) \mid u, w \in \Sigma_{sync}^* \text{ and } (x_1, x_2) \in \mathcal{I}\}.$$

We define its closure for language $L \subseteq \Sigma_{sync}^$: $\mathcal{C}^{\equiv_{\mathcal{I}}}(L) := \{u \in \Sigma_{sync}^* \mid \exists w \in L \text{ with } u \equiv_{\mathcal{I}} w\}$.*

► **Definition 5.2** (\mathcal{I} -closed global types). *Let \mathbf{G} be a global type \mathbf{G} . We say \mathbf{G} is \mathcal{I} -closed if $\mathcal{L}_{\text{fin}}(\text{GAut}(\mathbf{G})) = \mathcal{C}^{\equiv_{\mathcal{I}}}(\mathcal{L}_{\text{fin}}(\text{GAut}(\mathbf{G})))$.*

Note that \mathcal{I} -closedness is defined on the state machine $\text{GAut}(\mathbf{G})$ of \mathbf{G} with alphabet Σ_{sync} and not on its semantics $\mathcal{L}(\mathbf{G})$ with alphabet Σ_{async} .

► **Example 5.3.** The global type \mathbf{G}_{2BP} is \mathcal{I} -closed. Buyer a is involved in every interaction. Thus, for every two consecutive interactions, there is a role that is involved in both.

► **Algorithm 1** (Checking if \mathbf{G} is \mathcal{I} -closed). *Let \mathbf{G} be a global type. We construct the state machine $\text{GAut}(\mathbf{G})$. We need to check every consecutive occurrence of elements from Σ_{sync} for words from $\mathcal{L}(\text{GAut}(\mathbf{G}))$. For binder states, incoming and outgoing transition labels are always ε . This is why we slightly modify the state machine but preserve its language. We remove all variable states and rebend their only incoming transition to the state their only outgoing transition leads to. In addition, we merge binder states with their only successor. For every state q of this modified state machine, we consider the labels $x, y \in \Sigma_{sync}$ of every combination of incoming and outgoing transition of q . We check if $x \equiv_{\mathcal{I}} y$. If this is true for all x and y , we return true. If not, we return false.*

► **Lemma 5.4.** *A global type \mathbf{G} is \mathcal{I} -closed iff Algorithm 1 returns true.*

The proof can be found in the technical report [56]. This shows that the presented algorithm can be used to check \mathcal{I} -closedness. The algorithm considers every state and all combinations of transitions leading to and from it.

► **Proposition 5.5.** *For global type \mathbf{G} , checking if \mathbf{G} is \mathcal{I} -closed is in $O(|\mathbf{G}|^2)$.*

The tree-like shape of $\mathbf{GAut}(\mathbf{G})$ might suggest that this check can be done in linear time. However, this example shows that recursion can lead to a quadratic number of checks.

► **Example 5.6.** Let us consider the following global type for some $n \in \mathbb{N}$.

$$\mu t. + \begin{cases} p \rightarrow q_0 : m_0. q_0 \rightarrow r_0 : m_0. r_0 \rightarrow s_0 : m_0. 0 \\ p \rightarrow q_1 : m_1. q_1 \rightarrow r_1 : m_1. r_1 \rightarrow s_1 : m_1. t \\ \vdots \\ p \rightarrow q_n : m_n. q_n \rightarrow r_n : m_n. r_n \rightarrow s_n : m_n. t \end{cases}$$

It is obvious that $(p \rightarrow q_i : m_i, q_i \rightarrow r_i : m_i) \notin \mathcal{I}$ and $(q_i \rightarrow r_i : m_i, r_i \rightarrow s_i : m_i) \notin \mathcal{I}$ for every i . Because of the recursion, we need to check if $(r_i \rightarrow s_i : m_i, p \rightarrow q_j : m_j)$ is in \mathcal{I} for every $0 \neq i \neq j$. This might lead to a quadratic number of checks.

If a global type \mathbf{G} is \mathcal{I} -closed, we can apply the results for its \mathcal{I} -closed HMSC encoding $H(\mathbf{G})$, for which checking implementability is in PSPACE. With Corollary 4.11, the projection by erasure implements \mathbf{G} .

► **Corollary 5.7.** *Checking implementability of 0-reachable, \mathcal{I} -closed global types with sender-driven choice is in PSPACE.*

► **Example 5.8.** This implementable global type is not \mathcal{I} -closed: $p \rightarrow q : m. r \rightarrow s : m. 0$.

Detecting Non-local Choice in HMSCs

For HMSCs, there are no restrictions on branching. Similar to choice for global types, the idea of imposing restrictions on choice was studied for HMSCs [8, 50, 48, 34, 31]. We refer to [45] for an overview. Here, we focus on results that seem most promising for developing algorithms to check implementability of global types with better worst-case complexity. The work by Dan et al. [26] centers around the idea of non-local choice. Intuitively, non-local choice yields scenarios that make it impossible to implement the language. In fact, if a language is not implementable, there is some non-local choice. Thus, checking implementability amounts to checking non-local choice freedom. For this definition, they showed insufficiency of Baker's condition [6] and reformulated the closure conditions for safe realisability by Alur et al. [2]. In particular, they provide a definition that is based on projected words of a language in contrast to explicit choice. While it is straightforward to check their definition for finite collections of k BMSCs with n events in $O(k^2 \cdot |\mathcal{P}| + n \cdot |\mathcal{P}|)$, it is unclear how to check their condition for languages with infinitely many elements. The design of such a check is far from trivial as their definition does not give any insight about local behaviour and their algorithm heavily relies on the finite nature of finite collections of BMSCs.

Payload Implementability

A deadlock-free CSM implements a global type if their languages are precisely the same. In the HMSC domain, a variant of the implementability problem has been studied. Intuitively, it allows to add fresh data to the payload of an existing message and protocol fidelity allows to omit the additional payload data. This allows to add synchronisation messages to existing

interactions and can make unimplementable global types implementable while preserving the structure of the protocol. It can also be used if a global type is rejected by a projection operator or the run time of the previous algorithms is not acceptable.

► **Definition 5.9** (Payload implementability). *Let L be a language with message alphabet \mathcal{V}_1 . We say that L is payload implementable if there is a message alphabet \mathcal{V}_2 for a deadlock-free CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ with A_p over $\{p \triangleright q!m, p \triangleleft q?m \mid q \in \mathcal{P}, m \in \mathcal{V}_1 \times \mathcal{V}_2\}$ such that its language is the same when projecting onto the message alphabet \mathcal{V}_1 , i.e., $\mathcal{C}^\sim(L) = \mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}) \downarrow_{\mathcal{V}_1}$, where $(p \triangleright q!(m_1, m_2)) \downarrow_{\mathcal{V}_1} := p \triangleright q!m_1$ and $(q \triangleleft p?(m_1, m_2)) \downarrow_{\mathcal{V}_1} := q \triangleleft p?m_1$ and is lifted to words and languages as expected.*

The finite language $\mathcal{L}_{\text{fin}}(H)$ of a local HMSC H is always payload implementable with a deadlock-free CSM of linear size.

► **Definition 5.10** (Local HMSCs [31]). *Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. We call H local if $\mu(v^I)$ has a unique minimal event and there is a function $\text{root}: V \rightarrow \mathcal{P}$ such that for every $(v, u) \in E$, it holds that $\mu(u)$ has a unique minimal event e and e belongs to $\text{root}(v)$, i.e., for $\mu(u) = (N, p, f, l, (\leq_p)_{p \in \mathcal{P}})$, we have that $p(e) = \text{root}(v)$ and $e \leq e'$ for every $e' \in N$.*

► **Proposition 5.11** (Prop. 21 [31]). *For any local HMSC H , $\mathcal{L}_{\text{fin}}(H)$ is payload implementable.*

With Lemma 4.10, we can use the implementation of a local $H(\mathbf{G})$ for a 0-reachable global type \mathbf{G} .

► **Corollary 5.12.** *Let \mathbf{G} be a 0-reachable for which $H(\mathbf{G})$ is local. Then, \mathbf{G} can be implemented with a CSM of linear size.*

The algorithm to construct a deadlock-free CSM [31, Sec. 5.2] suggests that the BMSCs for such HMSCs need to be maximal – in the sense that any vertex with a single successor is collapsed with its successor. If this was not the case, the result would claim that the language of the following global type is payload implementable: $\mu t. + \begin{cases} p \rightarrow q: m_1. r \rightarrow s: m_2. t \\ p \rightarrow q: m_3. 0 \end{cases}$. However, it is easy to see that it is not payload implementable since there is no interaction between p , which decides whether to stay in the loop or not, and r . Thus, we cannot simply check whether $H(\mathbf{G})$ is local. In fact, it would always be. Instead, we first need to minimise it and then check whether it is local. If we collapse the two consecutive vertices with independent pairs of roles in this example, the HMSC is not local. The representation of the HMSC matters which shows that local as property is rather a syntactic than a semantic notion.

► **Algorithm 2** (Checking if $H(\mathbf{G})$ is local – directly on $\text{GAut}(\mathbf{G})$). *Let \mathbf{G} be a global type \mathbf{G} . We consider the finite trace w' of every longest branch-free, loop-free and non-initial run in the state machine $\text{GAut}(\mathbf{G})$. We split the (synchronous) interactions into asynchronous events: $w = \text{split}(w') = w_1 \dots w_n$. We need to check if there is $u \sim w$ with $u = u_1 \dots u_n$ such that $u_1 \neq w_1$. For this, we can construct an MSC for w [30, Sec. 3.1] and check if there is a single minimal event. This works because MSCs are closed under \sim [57, Lm. 5]. If the MSC of every trace w' has a single minimal event, we return true. If not, we return false.*

It is straightforward that this mimics the corresponding check for the HMSC $H(\mathbf{G})$ and, with similar modifications as for Algorithm 1, the check can be done in $O(|\mathbf{G}|)$.

► **Proposition 5.13.** *For a global type \mathbf{G} , Algorithm 2 returns true iff $H(\mathbf{G})$ is local.*

Ben-Abdallah and Leue [8] introduced local-choice HMSCs, which are as expressive as local HMSCs. Their condition also uses a root-function and minimal events but quantifies over paths. Every local HMSC is a local-choice HMSC and every local-choice HMSC can be translated to a local HMSC that accepts the same language with a quadratic blow-up [31]. It is straightforward to adapt the Algorithm 2 to check if a global type is local-choice. If this is the case, we translate the protocol and use the implementation for the translated protocol.

6 Implementability with Intra-role Reordering

In this section, we introduce a generalisation of the implementability problem that relaxes the total event order for each role. We prove that this generalisation is undecidable in general.

A Case for More Reordering

From the perspective of a single role, each word in its language consists of a sequence of send and receive events. Choice in global types happens by sending (and not by receiving). Because of this, one can argue that a role should be able to receive messages from different senders in any order between sending two messages. In practice, receiving a message can induce a task with non-trivial computation that our model does not account for. Therefore, such a reordering for a sequence of receive events can have outsized performance benefits. In addition, there are global types that can be implemented with regard to this generalised relation even if no (standard) implementation exists.

► **Example 6.1** (Example for intra-role reordering). Let us consider a global type where a central coordinator p distributes independent tasks to different roles in rounds:

$$\mathbf{G}_{\text{TC}} := \mu t. \begin{cases} p \rightarrow q_1 : \text{task} \dots p \rightarrow q_n : \text{task} \cdot q_1 \rightarrow p : \text{result} \dots q_n \rightarrow p : \text{result} \cdot t \\ p \rightarrow q_1 : \text{done} \dots p \rightarrow q_n : \text{done} \cdot 0 \end{cases}$$

Since all tasks in each round are independent, p can benefit from receiving the results in the order they arrive instead of busy-waiting.

We generalise the indistinguishability relation \sim accordingly.

► **Definition 6.2** (Intra-role indistinguishability relation \approx). We define a family of intra-role indistinguishability relations $\approx_i \subseteq \Sigma^* \times \Sigma^*$, for $i \geq 0$ as follows. For all $w, u \in \Sigma^*$, $w \sim_i u$ entails $w \approx_i u$. For $i = 1$, we define: if $q \neq r$, then $w.p \triangleleft q ? m.p \triangleleft r ? m'.u \sim_1 w.p \triangleleft r ? m'.p \triangleleft q ? m.u$. Based on this, we define \approx analogously to \sim . Let w, w', w'' be words s.t. $w \approx_1 w'$ and $w' \approx_i w''$ for some i . Then $w \approx_{i+1} w''$. We define $w \approx u$ if $w \approx_n u$ for some n . It is straightforward that \approx is an equivalence relation. Define $u \preceq_{\approx} v$ if there is $w \in \Sigma^*$ such that $u.w \approx v$. Observe that $u \sim v$ iff $u \preceq_{\approx} v$ and $v \preceq_{\approx} u$. We extend \approx to infinite words and languages as for \sim .

► **Definition 6.3** (Implementability w.r.t. \approx). A global type \mathbf{G} is implementable with regard to \approx if there exists a deadlock-free CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ such that (i) $\mathcal{L}(\mathbf{G}) \subseteq \mathcal{C}^{\approx}(\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}))$ and (ii) $\mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G})) = \mathcal{C}^{\approx}(\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}))$. We say that $\{\{A_p\}_{p \in \mathcal{P}}\} \approx$ -implements \mathbf{G} .

In this section, we emphasise the indistinguishability relation, e.g., \approx -implementable. We could have also followed the definition of \sim -implementability and required $\mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G})) = \mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\})$. This, however, requires the CSM to be closed under \approx . In general, this might not be possible with finitely many states. In particular, if there is a loop without any send events for a role, the labels in the loop would introduce an infinite closure if we require that $\mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G})) \downarrow_{\Sigma_p} = \mathcal{L}(A_p)$.

► **Example 6.4**. We consider a variant of \mathbf{G}_{TC} from Example 6.1 with $n = 2$ where q_1 and q_2 send a log message to r after receiving the task and before sending the result back:

$$\mathbf{G}_{\text{TCLog}} := \mu t. \begin{cases} p \rightarrow q_1 : \text{task} \cdot p \rightarrow q_2 : \text{task} \cdot q_1 \rightarrow r : \text{log} \cdot q_2 \rightarrow r : \text{log} \cdot q_1 \rightarrow p : \text{result} \cdot q_2 \rightarrow p : \text{result} \cdot t \\ p \rightarrow q_1 : \text{done} \cdot p \rightarrow q_2 : \text{done} \cdot 0 \end{cases}$$

There is no FSM for r that precisely accepts $\mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G}_{\text{TCLog}})) \downarrow_{\Sigma_r}$. If we rely on the fact that q_1 and q_2 send the same number of log-messages to r , we can use an FSM A_r with a single state (both initial and final) with two transitions: one for the log-message from q_1 and q_2

each, that lead back to the only state. For this, it holds that $\mathcal{C}^{\approx}(\mathcal{L}(\mathbf{G}_{\text{TCLog}})) \downarrow_{\Sigma_r} \subseteq \mathcal{L}(A_r)$. If we cannot rely on this, the FSM would need to keep track of the difference, which can be unbounded and thus not recognisable by an FSM.

This is why we chose a more permissive definition which is required to cover at least as much as specified in the global type (i) and the \approx -closure of both are the same (ii).

It is trivial that any \sim -implementation for a global type does also \approx -implement it.

► **Proposition 6.5.** *Let \mathbf{G} be a global type that is \sim -implemented by the CSM $\{\{A_p\}\}_{p \in \mathcal{P}}$. Then, $\{\{A_p\}\}_{p \in \mathcal{P}}$ also \approx -implements \mathbf{G} .*

For instance, the erasure candidate implementation is a \sim -implementation as well as a \approx -implementation for the task coordination protocol \mathbf{G}_{TC} from Example 6.1. Still, \approx -implementability gives more freedom and allows to consider all possible combinations of arrivals of results. In addition, \approx -implementability renders some global types implementable which would not be otherwise. For instance, those with a role that would need to receive different sequences, related by \approx though, in different branches it cannot distinguish (yet).

► **Example 6.6** (\approx -implementable but not \sim -implementable). Let us consider the following global type: $(p \rightarrow q : l. p \rightarrow r : m. q \rightarrow r : m. 0) + (p \rightarrow q : r. q \rightarrow r : m. p \rightarrow r : m. 0)$. This cannot be \sim -implemented because r would need to know about the choice to receive the messages from p and q in the correct order. However, it is \approx -implementable. The FSMs for p and q can be obtained with projection by erasure. For r , we can have an FSM that only accepts $r \triangleleft p ? m. r \triangleleft q ? m$ but also an FSM which accepts $r \triangleleft q ? m. r \triangleleft p ? m$ in addition. Note that r does not learn the choice in the second FSM even if it branches. Hence, it would not be implementable if it sent different messages in both branches later on. However, it could still learn by receiving and, afterwards, send different messages.

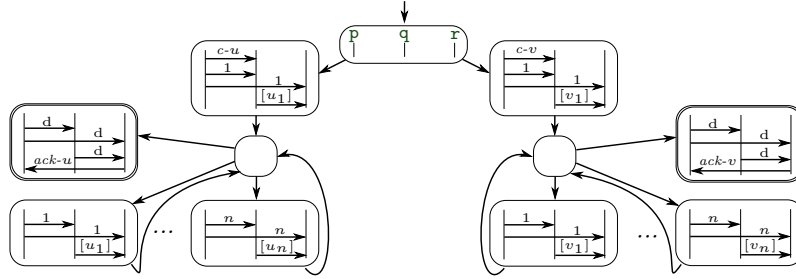
Implementability with Intra-role Reordering is Undecidable

Unfortunately, checking implementability with regard to \approx for global types (with directed choice) is undecidable. Intuitively, the reordering allows roles to drift arbitrarily far apart as the execution progresses which makes it hard to learn which choices were made.

We reduce the *Post Correspondence Problem* (PCP) [53] to the problem of checking implementability with regard to \approx . An instance of PCP over an alphabet Δ , $|\Delta| > 1$, is given by two finite lists (u_1, u_2, \dots, u_n) and (v_1, v_2, \dots, v_n) of finite words over Δ , also called tile sets. A solution to the instance is a sequence of indices $(i_j)_{1 \leq j \leq k}$ with $k \geq 1$ and $1 \leq i_j \leq n$ for all $1 \leq j \leq k$, such that $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$. To be precise, we present a reduction from the modified PCP (MPCP) [55, Sec. 5.2], which is also undecidable. It simply requires that a match starts with a specific pair – in our case we choose the pair with index 1. It is possible to directly reduce from PCP but the reduction from MPCP is more concise. Intuitively, we require that the solution starts with the first pair so there exists no trivial solution and choosing a single pair is more concise than all possible ones. Our encoding is the following global type where $x \in \{u, v\}$, $[x_i]$ denotes a sequence of message interactions with message $x_i[1], \dots, x_i[k]$ each for x_i of length k , message $c-x$ indicates choosing tile set x , and message $ack-x$ indicates acknowledging the tile set x :

$$\mathbf{G}_{\text{MPCP}} := + \begin{cases} G(u, r \rightarrow p : ack-u. 0) \\ G(v, r \rightarrow p : ack-v. 0) \end{cases} \quad \text{with}$$

$$G(x, X) := p \rightarrow q : c-x. p \rightarrow q : 1. p \rightarrow r : 1. q \rightarrow r : [x_1]. \mu t. + \begin{cases} p \rightarrow q : 1. p \rightarrow r : 1. q \rightarrow r : [x_1]. t \\ \vdots \\ p \rightarrow q : n. p \rightarrow r : n. q \rightarrow r : [x_n]. t \\ p \rightarrow q : d. p \rightarrow r : d. q \rightarrow r : d. X \end{cases}$$



■ **Figure 7** HMSC encoding $H(\mathbf{G}_{\text{MPCP}})$ of the MPCP encoding.

The HMSC encoding $H(\mathbf{G}_{\text{MPCP}})$ is illustrated in Figure 7. Intuitively, r eventually needs to know which branch was taken in order to match $ack-x$ with $c-x$ from the beginning. However, it can only know if there is no solution to the MPCP instance. In the full proof in technical report [56], we show that \mathbf{G}_{MPCP} is \approx -implementable iff the MPCP instance has no solution.

► **Theorem 6.7.** *Checking implementability with regard to \approx for 0-reachable global types with directed choice is undecidable.*

This result carries over to HMSCs if we consider safe realisability with regard to \approx .

► **Definition 6.8** (Safe realisability with regard to \approx). *An HMSC H is said to be safely realisable with regard to \approx if there exists a deadlock-free CSM $\{\{A_p\}_{p \in \mathcal{P}}\}$ such that the following holds: (i) $\mathcal{L}(H) \subseteq \mathcal{C}^\approx(\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}))$ and (ii) $\mathcal{C}^\approx(\mathcal{L}(H)) = \mathcal{C}^\approx(\mathcal{L}(\{\{A_p\}_{p \in \mathcal{P}}\}))$.*

► **Corollary 6.9.** *Checking safe realisability with regard to \approx for HMSCs is undecidable.*

It is obvious that a terminal vertex is reachable from every vertex in $H(\mathbf{G}_{\text{MPCP}})$. In fact, the HMSC encoding for \mathbf{G}_{MPCP} also satisfies a number of channel restrictions. The HMSC $H(\mathbf{G}_{\text{MPCP}})$ is existentially 1-bounded, 1-synchronisable and half-duplex [57]. For details on these channel restrictions, we refer to work by Stutz and Zufferey [57, Sec. 3.1].

The MPCP encoding only works since receive events can be reordered unboundedly in an execution. If we amended the definition of \approx to give each receive event a budget that depletes with every reordering, this encoding would not be possible. We leave a detailed analysis for future work.

7 Related Work

In this section, we solely cover related work which was not discussed before.

Multiparty Session Types. Session types originate in process algebra and were first introduced by Honda et al. [35] for binary sessions. For systems with more than two roles, they have been extended to multiparty session types [37]. We explained MST frameworks with classical projection operators. Other approaches do not focus on projection but only apply ideas from MST without the need for global types [54, 43].

Completeness and Sender-driven Choice. Our decidability result applies to global types with sender-driven choice. To the best of our knowledge, the work by Castagna et al. [16] is the only one to attempt completeness for global types with sender-driven choice. However, their definition of completeness is “less demanding than other ones” [16, Abs.]. For one

global type, they accept different implementations that generate different sets of traces [16, Def. 4.1 and Sec. 5.3]. Their conditions, given as inference rules, are not effective and their algorithmically checkable conditions can only exploit local information to disambiguate choices. In contrast, Majumdar et al. [45] employ a global availability analysis but, as classical projection operator, it suffers from the shortcomings presented in this work. For a detailed overview of MST frameworks with sender-driven choice, we refer to their work [45]. The global types by Castellani et al. [17] specify send and receive events independently and allow to receive from different senders. Dagnino et al. [25] consider similar global types but each term requires to send to a single receiver and to receive from a single sender upon branching though.

On the Synchronous Implementability Problem. We could not find a reference that shows decidability of the implementability problem in a synchronous setting, i.e., without channels. Before giving a proof sketch, let us remark that there are global types that can be implemented synchronously but not asynchronously, e.g., $p \rightarrow q:l.r \rightarrow q:l.0 + p \rightarrow q:r.r \rightarrow q:r.0$ because q can force the right choice by r . We sketch how to prove decidability of the synchronous implementability problem for global types (with sender-driven choice). One defines the synchronous semantics of CSMs and HMSCs as expected. For global types, one uses the independence relation \mathcal{I} (Def. 5.1), which defines reasonable reorderings for synchronous events in a distributed setting, similar to the indistinguishability relation \sim . It is straightforward that the HMSC encoding $H(-)$ for global types [57] also works for the synchronous setting (cf. Thm. 4.4). Thus, every implementation for $H(\mathbf{G})$ is also an implementation for \mathbf{G} . For the asynchronous setting, we used [2, Thm. 13], which shows that the canonical candidate implementation implements an HMSC if it is implementable. Alur et al. [2, Sec. 8] also considered the synchronous setting. They observe that both Theorem 5 and 8, basis for Theorem 13, stay valid under these modified conditions. Together with our results, the erasure candidate implementation implements a global type if it is implementable. Because of the synchronous semantics, its state space is finite and can be model-checked against the global type, yielding a PSPACE-procedure for the synchronous implementability problem, and thus decidability. Closest are works by Jongmans and Yoshida [40] and Glabbeek et al. [62]. Jongmans and Yoshida consider quite restrictive synchronous semantics for global types [40, Ex. 3] that does not allow the natural reorderings in a distributed setting, as enabled by \mathcal{I} , e.g., $p \rightarrow q:m.r \rightarrow s:m.0$ (Ex. 5.8) is considered unimplementable. Glabbeek et al. [62] present a projection operator that is complete for various notions of lock-freedom, a typical liveness property, and investigate how much fairness is required for those.

Subtyping and MST Extensions. In this work, we do not distinguish between local types and implementations but use local types directly as implementations. Intuitively, subtyping studies how to give freedom in the implementation while preserving the correctness properties. The intra-role indistinguishability relation \approx , which allows to reorder receive events for a role, resembles subtyping to some extent, e.g., the work by Cutner et al. [24]. A detailed investigation of this relation is beyond the scope of this work. For details on subtyping, we refer to work by Chen et al. [22, 21], Lange and Yoshida [42], and Bravetti et al. [14]. Various extensions to make MST verification applicable to more scenarios were studied: for instance delegation [36, 37, 18], dependent session types [59, 29, 60], parametrised session types [20, 29], gradual session types [39], or dynamic self-adaption [33]. Context-free session types [58, 41] provide a more expressive way to specify protocols. Research on fault-tolerant MSTs [63, 7] investigates ways to weaken the strict assumptions about reliable channels.

Communicating State Machines. The connection of MSTs and CSMs was studied soon after MSTs had been proposed [28]. CSMs are known to be Turing-powerful [13]. Decidable classes have been obtained for different semantics, e.g., half-duplex communication for two roles [19], input-bounded [10], and unreliable/lossy channels [1], as well for restricted communication topology [52, 61]. Similar restrictions for CSMs are existential boundedness [30] and synchronisability [12, 32]. It was shown that global types can only express existentially 1-bounded, 1-synchronisable and half-duplex communication [57] while Bollig et al. [11] established a connection between synchronisability and MSO logic.

High-level Message Sequence Charts. Globally-cooperative HMSCs were independently introduced by Morin [49] as c-HMSCs. Their communication graph is weakly connected. The class of bounded HMSCs [4] requires it to be strongly connected. Historically, it was introduced before the class of globally-cooperative HMSCs and, after the latter has been introduced, safe realisability for bounded HMSCs was also shown to be EXPSPACE-complete [44]. This class was independently introduced as regular HMSCs by Muscholl and Peled [51]. Both terms are justified: the language generated by a *regular* HMSC is regular and every *bounded* HMSC can be implemented with universally bounded channels. In fact, a HMSC is bounded if and only if it is a globally-cooperative and it has universally bounded channels [31, Prop. 4].

8 Conclusion

We have proven decidability of the implementability problem for global types with generalised choice from MSTs – under the mild assumption that protocols can (but do not need to) terminate. To point at the origin for incompleteness of classical projection operators, we gave a visual explanation of the projection with various merge operators on finite state machines, which define the semantics of global and local types. To prove decidability, we formally related the implementability problem for global types with the safe realisability problem for HMSCs. While safe realisability is undecidable in general, we showed that implementable global types do always belong to the class of globally-cooperative HMSCs. There are global types that are outside of this class but the syntax of global types allowed us to prove that those cannot be implemented. Another key was the extension of the HMSC results to infinite executions. We also gave a comprehensive overview of MSC techniques and adapted some to the MST setting. Furthermore, we introduced a performance-oriented generalisation of the implementability problem which, however, we proved to be undecidable in general.

References

- 1 Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28 – July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318. Springer, 1998. doi:10.1007/BFb0028754.
- 2 Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Trans. Software Eng.*, 29(7):623–633, 2003. doi:10.1109/TSE.2003.1214326.
- 3 Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005. doi:10.1016/j.tcs.2004.09.034.
- 4 Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 1999. doi:10.1007/3-540-48320-9_10.

- 5 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniérou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/25000000031.
- 6 Paul Baker, Paul Bristow, Clive Jervis, David J. King, Robert Thomson, Bill Mitchell, and Simon Burton. Detecting and resolving semantic pathologies in UML sequence diagrams. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 50–59. ACM, 2005. doi:10.1145/1081706.1081716.
- 7 Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised multiparty session types with crash-stop failures. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPICs*, pages 35:1–35:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CONCUR.2022.35.
- 8 Hanène Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In Ed Brinksma, editor, *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS '97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volume 1217 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 1997. doi:10.1007/BFb0035393.
- 9 Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Asynchronous timed session types – from duality to time-sensitive processes. In Luís Caires, editor, *Programming Languages and Systems – 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 583–610. Springer, 2019. doi:10.1007/978-3-030-17184-1_21.
- 10 Benedikt Bollig, Alain Finkel, and Amrita Suresh. Bounded reachability problems are decidable in FIFO machines. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPICs*, pages 49:1–49:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.CONCUR.2020.49.
- 11 Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Étienne Lozes, and Amrita Suresh. A unifying framework for deciding synchronizability. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPICs*, pages 14:1–14:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CONCUR.2021.14.
- 12 Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the completeness of verifying message passing programs under bounded asynchrony. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification – 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 372–391. Springer, 2018. doi:10.1007/978-3-319-96142-2_23.
- 13 Daniel Brand and Pitro Zafriopulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. doi:10.1145/322374.322380.
- 14 Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018. doi:10.1016/j.tcs.2018.02.010.
- 15 Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Stephen Ross-Talbot. A theoretical basis of communication-centred concurrent programming, 2005.
- 16 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:24)2012.

- 17 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Asynchronous sessions with input races. In Marco Carbone and Romyana Neykova, editors, *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022*, volume 356 of *EPTCS*, pages 12–23, 2022. doi:10.4204/EPTCS.356.2.
- 18 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. Global types with internal delegation. *Theor. Comput. Sci.*, 807:128–153, 2020. doi:10.1016/j.tcs.2019.09.027.
- 19 Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Inf. Comput.*, 202(2):166–190, 2005. doi:10.1016/j.ic.2005.05.006.
- 20 Minas Charalambides, Peter Dinges, and Gul A. Agha. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Sci. Comput. Program.*, 115-116:100–126, 2016. doi:10.1016/j.scico.2015.10.006.
- 21 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.
- 22 Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. On the preciseness of subtyping in session types. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014*, pages 135–146. ACM, 2014. doi:10.1145/2643135.2643138.
- 23 Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A gentle introduction to multiparty asynchronous session types. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming – 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures*, volume 9104 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2015. doi:10.1007/978-3-319-18941-3_4.
- 24 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2–6, 2022*, pages 246–261. ACM, 2022. doi:10.1145/3503221.3508404.
- 25 Francesco Dagnino, Paola Giannini, and Mariangiola Dezani-Ciancaglini. Deconfined global types for asynchronous sessions. In Ferruccio Damiani and Ornela Dardha, editors, *Coordination Models and Languages – 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, volume 12717 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2021. doi:10.1007/978-3-030-78142-2_3.
- 26 Haitao Dan, Robert M. Hierons, and Steve Counsell. Non-local choice and implied scenarios. In José Luiz Fiadeiro, Stefania Gnesi, and Andrea Maggiolo-Schettini, editors, *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13-18 September 2010*, pages 53–62. IEEE Computer Society, 2010. doi:10.1109/SEFM.2010.14.
- 27 Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. Resource-aware session types for digital contracts. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16. IEEE, 2021. doi:10.1109/CSF51468.2021.00004.
- 28 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 – April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. doi:10.1007/978-3-642-28869-2_10.

- 29 Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Log. Methods Comput. Sci.*, 8(4), 2012. doi:10.2168/LMCS-8(4:6)2012.
- 30 Blaise Genest, Dietrich Kuske, and Anca Muscholl. On communicating automata with bounded channels. *Fundam. Inform.*, 80(1-3):147–167, 2007. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09>.
- 31 Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. *J. Comput. Syst. Sci.*, 72(4):617–647, 2006. doi:10.1016/j.jcss.2005.09.007.
- 32 Cinzia Di Giusto, Laetitia Laversa, and Étienne Lozes. On the k-synchronizability of systems. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures – 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12077 of *Lecture Notes in Computer Science*, pages 157–176. Springer, 2020. doi:10.1007/978-3-030-45231-5_9.
- 33 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty session types for safe runtime adaptation in an actor language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPICs*, pages 10:1–10:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ECOOP.2021.10.
- 34 Loïc Hélouët. Some pathological message sequence charts, and how to detect them. In Rick Reed and Jeanne Reed, editors, *SDL 2001: Meeting UML, 10th International SDL Forum Copenhagen, Denmark, June 27-29, 2001, Proceedings*, volume 2078 of *Lecture Notes in Computer Science*, pages 348–364. Springer, 2001. doi:10.1007/3-540-48213-X_22.
- 35 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.
- 36 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems – ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 – April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.
- 37 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. doi:10.1145/1328438.1328472.
- 38 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.
- 39 Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *J. Funct. Program.*, 29:e17, 2019. doi:10.1017/S0956796819000169.
- 40 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In Peter Müller, editor, *Programming Languages and Systems – 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12075 of *Lecture Notes in Computer Science*, pages 251–279. Springer, 2020. doi:10.1007/978-3-030-44914-8_10.
- 41 Alex C. Keizer, Henning Basold, and Jorge A. Pérez. Session coalgebras: A coalgebraic view on regular and context-free session types. *ACM Trans. Program. Lang. Syst.*, 44(3):18:1–18:45, 2022. doi:10.1145/3527633.

- 42 Julien Lange and Nobuko Yoshida. On the undecidability of asynchronous session subtyping. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures – 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017. doi:10.1007/978-3-662-54458-7_26.
- 43 Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification – 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019. doi:10.1007/978-3-030-25540-4_6.
- 44 Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theor. Comput. Sci.*, 309(1-3):529–554, 2003. doi:10.1016/j.tcs.2003.08.002.
- 45 Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising projection in asynchronous multiparty session types. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, volume 203 of *LIPICs*, pages 35:1–35:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.CONCUR.2021.35.
- 46 Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. Motion session types for robotic interactions (brave new idea paper). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, volume 134 of *LIPICs*, pages 28:1–28:27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ECOOP.2019.28.
- 47 Sjouke Mauw and Michel A. Reniers. High-level message sequence charts. In Ana R. Cavalli and Amardeo Sarma, editors, *SDL '97 Time for Testing, SDL, MSC and Trends – 8th International SDL Forum, Evry, France, 23-29 September 1997, Proceedings*, pages 291–306. Elsevier, 1997.
- 48 Arjan J. Mooij, Nicolae Goga, and Judi Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2005. doi:10.1007/978-3-540-31984-9_21.
- 49 Rémi Morin. Recognizable sets of message sequence charts. In Helmut Alt and Afonso Ferreira, editors, *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes – Juan les Pins, France, March 14-16, 2002, Proceedings*, volume 2285 of *Lecture Notes in Computer Science*, pages 523–534. Springer, 2002. doi:10.1007/3-540-45841-7_43.
- 50 Henry Muccini. Detecting implied scenarios analyzing non-local branching choices. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2621 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2003. doi:10.1007/3-540-36578-8_26.
- 51 Anca Muscholl and Doron A. Peled. Message sequence graphs and decision problems on mazurkiewicz traces. In Mirosław Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*, volume 1672 of *Lecture Notes in Computer Science*, pages 81–91. Springer, 1999. doi:10.1007/3-540-48340-3_8.
- 52 Wuxu Peng and S. Purushothaman. Analysis of a class of communicating finite state machines. *Acta Informatica*, 29(6/7):499–522, 1992. doi:10.1007/BF01185558.
- 53 Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.
- 54 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.

- 55 Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.
- 56 Felix Stutz. Asynchronous multiparty session type implementability is decidable – lessons learned from message sequence charts, 2023. doi:10.48550/ARXIV.2302.11272.
- 57 Felix Stutz and Damien Zufferey. Comparing channel restrictions of communicating state machines, high-level message sequence charts, and multiparty session types. In Pierre Ganty and Dario Della Monica, editors, *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022*, volume 370 of *EPTCS*, pages 194–212, 2022. doi:10.4204/EPTCS.370.13.
- 58 Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 462–475. ACM, 2016. doi:10.1145/2951913.2951926.
- 59 Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 161–172. ACM, 2011. doi:10.1145/2003476.2003499.
- 60 Bernardo Toninho and Nobuko Yoshida. Depending on session-typed processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures – 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10803 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2018. doi:10.1007/978-3-319-89366-2_7.
- 61 Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Context-bounded analysis of concurrent queue systems. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2008. doi:10.1007/978-3-540-78800-3_21.
- 62 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make session types complete for lock-freedom. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 – July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470531.
- 63 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485501.
- 64 Nobuko Yoshida and Lorenzo Gheri. A very gentle introduction to multiparty session types. In Dang Van Hung and Meenakshi D’Souza, editors, *Distributed Computing and Internet Technology – 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, volume 11969 of *Lecture Notes in Computer Science*, pages 73–93. Springer, 2020. doi:10.1007/978-3-030-36987-3_5.
- 65 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing – 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs

Felix Suchert  

TU Dresden, Germany

Lisza Zeidler 

Barkhausen Institut, Dresden, Germany

Jeronimo Castrillon  

TU Dresden, Germany

Sebastian Ertel¹  

Barkhausen Institut, Dresden, Germany

Abstract

SAT/SMT-solvers and model checkers automate formal verification of sequential programs. Formal reasoning about scalable concurrent programs is still manual and requires expert knowledge. But scalability is a fundamental requirement of current and future programs.

Sequential imperative programs compose statements, function/method calls and control flow constructs. Concurrent programming models provide constructs for concurrent composition. Concurrency abstractions such as threads and synchronization primitives such as locks compose the individual parts of a concurrent program that are meant to execute in parallel. We propose to rather compose the individual parts again using sequential composition and compile this sequential composition into a concurrent one. The developer can use existing tools to formally verify the sequential program while the translated concurrent program provides the dearly requested scalability.

Following this insight, we present *ConDRust*, a new programming model and compiler for Rust programs. The *ConDRust* compiler translates sequential composition into a concurrent composition based on threads and message-passing channels. During compilation, the compiler preserves the semantics of the sequential program along with much desired properties such as determinism.

Our evaluation shows that our *ConDRust* compiler generates concurrent deterministic code that can outperform even non-deterministic programs by up to a factor of three for irregular algorithms that are particularly hard to parallelize.

2012 ACM Subject Classification Theory of computation → Parallel computing models; Software and its engineering → Parallel programming languages

Keywords and phrases concurrent programming, verification, scalability

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.33

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.16>

Funding *Felix Suchert:* was funded by the EU Horizon 2020 Programme under grant agreement No 957269 (EVEREST).

Lisza Zeidler: was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 469256231.

Acknowledgements The authors would like to thank the anonymous reviewers for their invaluable feedback in the submission process.

¹ Corresponding author



© Felix Suchert, Lisza Zeidler, Jeronimo Castrillon, and Sebastian Ertel; licensed under Creative Commons License CC-BY 4.0

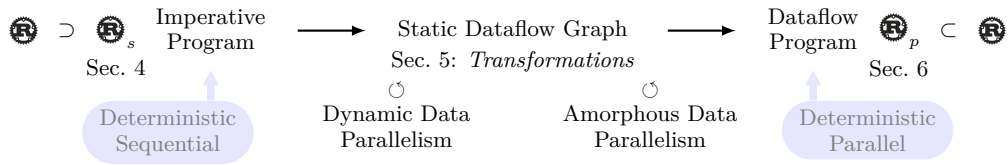
37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 33; pp. 33:1–33:39



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany





■ **Figure 1** The *ConDRust* compiler translates imperative programs written in \mathbb{R}_s , a subset of Rust for sequential composition, into dataflow programs in \mathbb{R}_p , a subset of Rust for parallel composition.

1 Introduction

Formal verification of sequential programs can be automated to a large extent which makes it ready for widespread adoption. Verification of concurrent multi-core shared-memory programs, instead, can only be automated to some extent and requires expert knowledge. This is a major hurdle for safe systems which must rely on scalable parallelism to overcome the physical boundary in current and future processors.

A formally verified program carries mathematical proof that certain properties of the program hold. Determinism is such an interesting property. Deterministic programs are straightforward to debug [12]. A deterministic execution increases the time predictability of IoT systems [50] and establishes latency boundaries of service-level agreements in the cloud [20, 4]. In recent database management systems with transaction support, determinism removes costly synchronization and challenging distributed failure scenarios [52, 40].

Formal verification of program properties, such as determinism, proceeds along two directions. Proof assistants such as Coq allow expressing properties in higher-order logic but require a manual proof from the developer. SAT/SMT-based verifiers such as Prusti and model checkers such as Kani for Rust programs are restricted to properties formulated in first-order logic but calculate the proof automatically [3, 54]. That is, the hard part of formal verification is automated which makes them particularly interesting for widespread adoption. But concurrent programs require separation logic to state and prove their properties [51, 43]. Encoding separation logic into first-order logic is still ongoing research [15, 14, 44, 43] and needs to sacrifice important (higher-order) parts. Expert knowledge in Coq is required to take full advantage of separation logic [30]. At the time of this writing, none of the formal verification tools for Rust programmers supports reasoning about concurrent programs.

Our insight is that two main steps are needed to translate a sequential program into a concurrent one. We call these steps *Decompose* and *Recompose*. The Decompose step breaks the sequential composition of a program to create independent parts that can execute in parallel. The Recompose step composes these parts again using concurrency abstractions such as threads and synchronization primitives such as locks or message-passing. We refer to this as *concurrent composition*. In programming models such as threads with locks, message-passing or software transactional memory (STM), the developer has to perform both steps manually. Automatic approaches to tackle both steps require a precise points-to analysis to create a concurrent program without concurrency hazards such as data races or deadlocks. This analysis is known to be undecidable in general [48]. Hence, researchers resort to speculative approaches [17] or language constraints for a precise dependence analysis [6].

The *ConDRust* approach

In this paper, we propose *ConDRust*, a new *sequential* programming model for the *concurrent composition* in the Recompose phase. Our *ConDRust* compiler translates a sequential composition into a concurrent one automatically. This allows for testing and formal verification to be performed on the sequential program, with guarantees that are carried into the concurrent

one. More specifically, our current prototype compiler supports a well-defined subset of *safe* Rust for sequential composition and generates concurrent dataflow composition in a well-defined subset of *safe* Rust with threads and message-passing. The dataflow execution model is the runtime representation for scalable parallelism in many domains such as embedded systems, database systems and machine learning frameworks [38, 23, 27, 58]. Our compiler design is based on rewriting steps that preserve the semantics of the sequential input program. This includes the semantics of control flow constructs such as loops but also properties such as determinism. Formally verifying the compiler is a larger effort [39] that is beyond the scope of this paper. This paper, instead, investigates to what extent the programming model is applicable to real-world programs and whether the compiler can generate scalable concurrent composition that is at least on par with existing concurrent programming models. Throughout the paper, we point to novel and interesting research directions that our approach introduces.

Concretely, we make the following contributions:

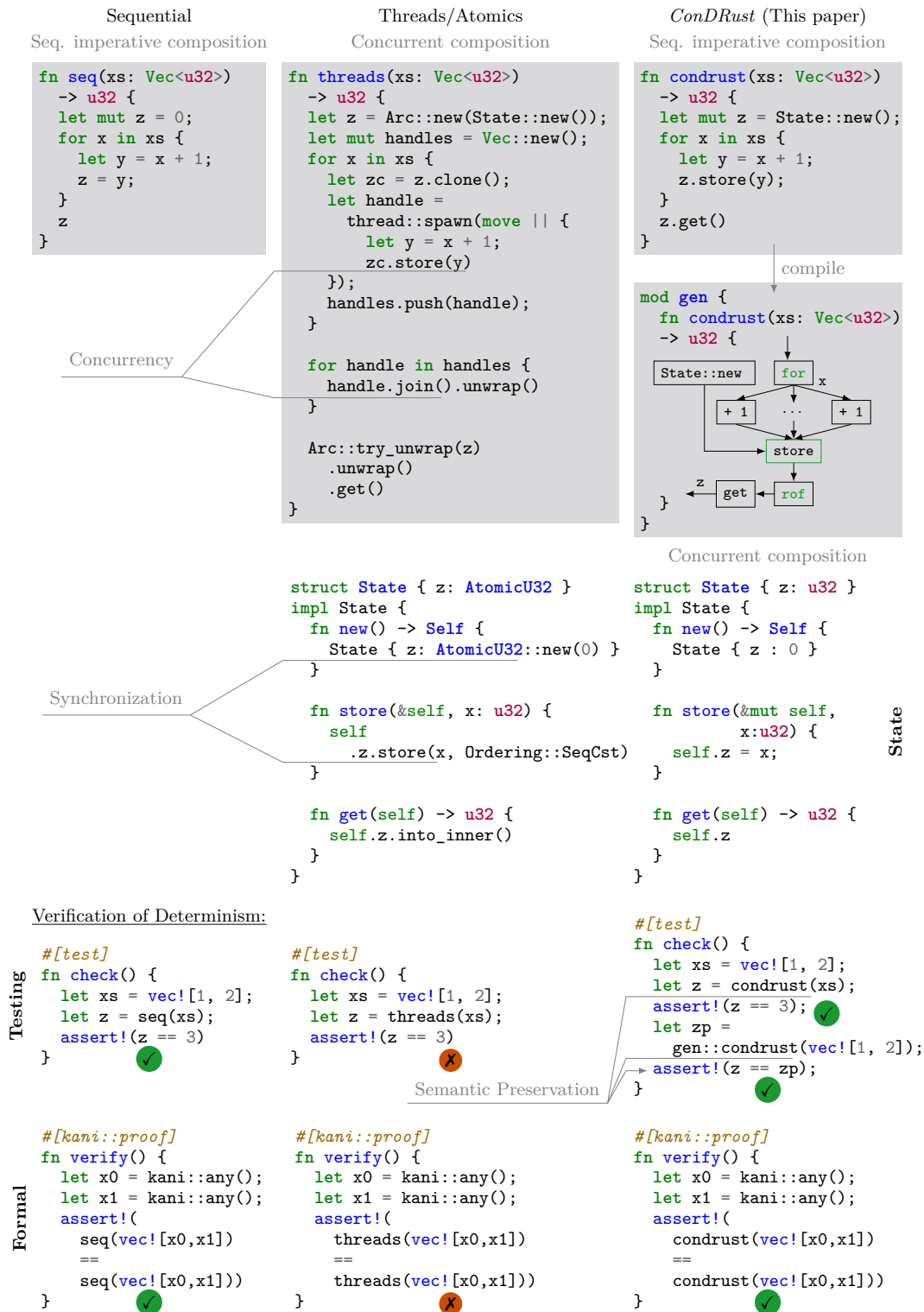
- The main contribution of this paper is a new programming model and compiler for the compositional fragment, i.e., subset, of safe Rust (Sections 2 and 3).
- We define \mathbb{R}_s , a subset of Rust for sequential imperative composition with abstractions, calls, variables but without references, and formally specify its type system and operational semantics (in Section 4 and the Appendix).
- We formally specify \mathbb{R}_p , the subset of Rust for concurrent composition that the *ConDRust* compiler targets in Section 6. The appendix contains the operational semantics, type system and a proof sketch for the deterministic execution.
- The key part of our compiler, visualized in Figure 1, is a transformation of sequential imperative \mathbb{R}_s composition into a functional representation based on the well-defined concept of state threads [35]. Our compiler lowers this functional representation into dataflow, a well-established abstraction for parallel execution [5, 2, 37, 29] (Section 5.1).
- Key to scalable concurrent composition are two transformations to exploit data parallelism even in stateful applications with (tail) recursion (Sections 5.2 and 5.3).
- Our current *ConDRust* prototype compiler consists of ca. 20K lines of Haskell code that can be lifted to Coq in future work to formally-verify semantic preservation (Section 7).
- To provide a fair baseline with the same guarantees, we re-implemented a deterministic STM (DSTM) algorithm [49] for an existing STM implementation in Rust. We ported 7 benchmarks from 3 different benchmark suites and provide implementations for sequential, threads/DSTM, threads/STM and *ConDRust* (ca. 12K lines of Rust code). Our evaluation in Section 8 shows that *ConDRust* produces programs that outperform all threads/DSTM and even some of the non-deterministic threads/STM programs by up to a factor of 3. We highlight directions for future work whenever *ConDRust* programs do not scale.

We review related work in Section 9 and conclude in Section 10.

2 Programming for Scalability: Decompose and Recompose

Before we introduce our programming model in more detail, we reflect on concurrent programming. We argue that concurrent programming and our *ConDRust* programming model contain the same *Decompose–Recompose* steps. That is, the reasoning for the developer to prepare a scalable concurrent program is the same as for writing a *ConDRust* program. But the implications are different. Figure 2 uses an inarguably contrived but easy to follow example for a side-by-side comparison. The left column shows the sequential program. The middle column lists the program with threads and atomics, i.e., transactions on a single variable. And the right column shows the sequential *ConDRust* program.

33:4 ConDRust: Scalable Deterministic Concurrency from Verifiable Rust Programs



■ **Figure 2** Comparison between sequential, concurrent and *ConDRust* programs and their properties. The sequential program executes deterministically and is amenable to verification. The concurrent program offers parallel speedup but compromises determinism and verifiability. The *ConDRust* approach preserves determinism and verifiability and compiles the program into a concurrent dataflow for scalable parallel execution.

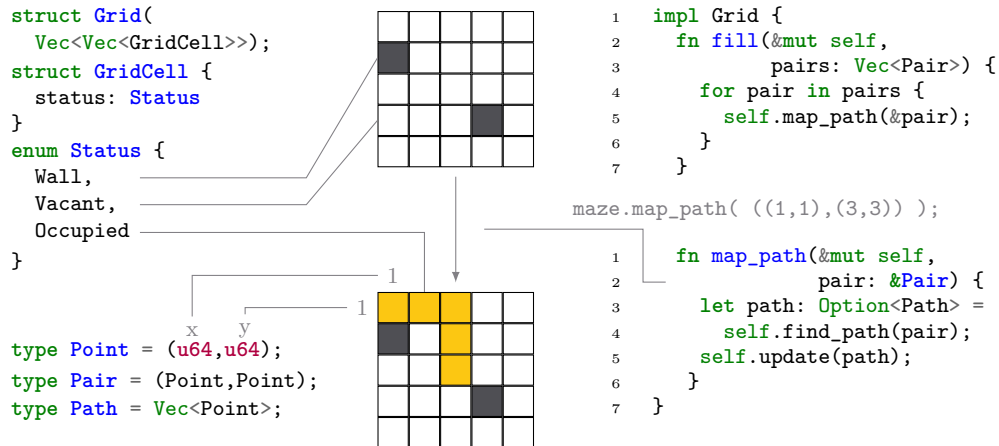
The sequential program on the left iterates over a vector of numbers `xs`. For each number, the program computes its increment and assigns it to a shared state `z`. The resulting value of `z` is always the increment of the last number in `xs`. This is by definition of the sequential execution order of statements and loop iterations in Rust. As such, we can check deterministic execution with a simple test case for a vector with 2 elements and let Kani formally verify this property in a simple proof harness (see bottom of left column in Figure 2). Apart from detecting potential overflow, which is inconsequential for this example, Kani succeeds.

To arrive at a scalable concurrent version of the program, the developer needs two steps. In the *Decompose* step, the developer identifies the parts of the program that can/should be executed in parallel and the state that needs to be shared. In our example, the developer selected the body of the loop for parallel execution. Any approach, that seeks to automate this step, needs to solve the thread granularity problem [47, 1]. Tools now assist in identifying the state [21]. The focus of this paper is on the second step. In the *Recompose* step, the developer replaces the sequential composition with a *concurrent composition*, e.g., for the loop iterations. Prevalent concurrent programming models for imperative programs consist of two parts for composition: *concurrency abstractions* and *synchronization primitives* to access (shared) state. The concurrent version of the program spawns a thread for each computation on the elements of the input vector `xs`. Thereby, it removes the sequential execution order of the `for` loop. Accesses to the state variable `z` now need to be protected with atomic operations (or transactions) to prevent data races. But this protection cannot recover the deterministic update order on `z`, even when opting for the strongest and least performing memory ordering: sequential consistency (`SeqCst`). The test's post-condition now may see `z == 2`. That is, the determinism property of the program is lost. Even worse, Kani cannot even help to detect this flaw because verification of concurrent programs needs a higher-order (separation) logic which is (currently) out of reach for model checkers like Kani².

To construct a *ConDRust* version of the program, the developer has to follow the same two steps. The *Decompose* step is the same as in the construction of the concurrent version. The developer identifies independent parts and shared state. In the *Recompose* step, the developer uses the sequential composition constructs of the host language such as for example statements, `for` loops, function calls and (imperative) method calls. The *ConDRust* program defines the update to the state `z` with a method rather than an assignment. Assignments are not supported in this paper because we tried to keep Θ_s minimal but can be added easily. The program is free of concurrency abstractions and synchronization primitives. The deterministic property on the result of the program can be tested and formally verified in the same way as for the sequential version. Note that formal verification is performed on the sequential `condrust` program. The *ConDRust* compiler is aware of the semantics of the composition constructs and preserves them during compilation. The generated dataflow graph (in `gen::condrust`) exhibits data parallelism for the computation of the increments and pipeline-parallelizes this with the update of the state `z`. The state update order is preserved. We can then dynamically verify that the generated concurrent program `gen::condrust` preserves the semantics, including the determinism property.

Concurrency vs. Parallelism. Throughout the paper, we use the words *concurrent* and *parallel* in the following well-established sense [53]. Concurrency means interleaved execution of computations. Parallelism exploits additional (multi-core) hardware to execute computations simultaneously. That is, concurrency does not necessarily imply parallelism. But in this

² <https://model-checking.github.io/kani/rust-feature-support.html#concurrency>



■ **Figure 3** Illustration of the labyrinth benchmark on a 2D grid in imperative sequential Rust.

paper, we focus on scalable concurrency. Scalable concurrency assumes multi-core hardware to turn explicit concurrency in the program into implicit parallelism. Hence, whenever we refer to parallelism, we mean independent concurrent computations in the program. ◀

3 The *ConDRust* programming model

In this section, we present the *ConDRust* programming model that we formally specify and embed into the Rust programming language (see Section 4). We compare programming in *ConDRust* to concurrent programming with threads and software transactional memory (STM). As a running example, we use Labyrinth, an irregular application from the STAMP benchmark suite [41]. Irregular programs contain algorithms where concurrent programming models particularly shine because such algorithms are notoriously hard to parallelize at compile-time. We start from the sequential version of the labyrinth algorithm and then develop a version based on threads and STM to compare it against programming in *ConDRust*.

3.1 The Labyrinth benchmark

The labyrinth benchmark implements Lee’s algorithm to find wire-paths between points on a multi-layer printed circuit board [36]. The challenge consists in finding paths that do not overlap across layers. The original STAMP implementation and our re-implementation in Rust execute on a 3D board (or 3D grid). In this section, we restrict ourselves to a 2D grid because it is easier to visualize while retaining the core principles required by our exposition.

Figure 3 illustrates the problem and shows a sequential imperative Rust implementation. The left-hand side defines a grid as a vector of vectors where a grid cell holds one of the three states: wall (in dark grey), vacant (in white) or occupied (in yellow). A point in the grid is a tuple of an x- and a y-coordinate. A pair is a tuple of two points and a path is a sequence of points represented as a vector. The algorithm to `fill` a given grid is given on the right-hand side of the figure. For each pair, the function `map_path` finds a path and declares the corresponding grid cells as occupied. Note that both steps `find_path` and `update` require access to the grid. That is what makes introducing concurrency particularly challenging for the developer.

3.2 Concurrent Labyrinth

Concurrently mapping paths onto the grid requires changes to the algorithm. Two candidate paths computed concurrently may overlap and thus one of them has to be re-computed. We visualize this effect in the middle column of Figure 4 and compare the threads/STM version in the left column with the *ConDRust* version in the right column. These alternative implementations are discussed in the following.

3.2.1 Threads/STM

We use threads for concurrency and synchronize grid access via STM. We chose STM over locks for two reasons. First, STM guarantees data-race freedom without creating deadlocks. Second, deterministic STM algorithms exist that provide the same deterministic execution properties as *ConDRust*. For this paper, we used `rust-stm`³, an STM implementation in Rust, that follows the design of the STM in Haskell [28].

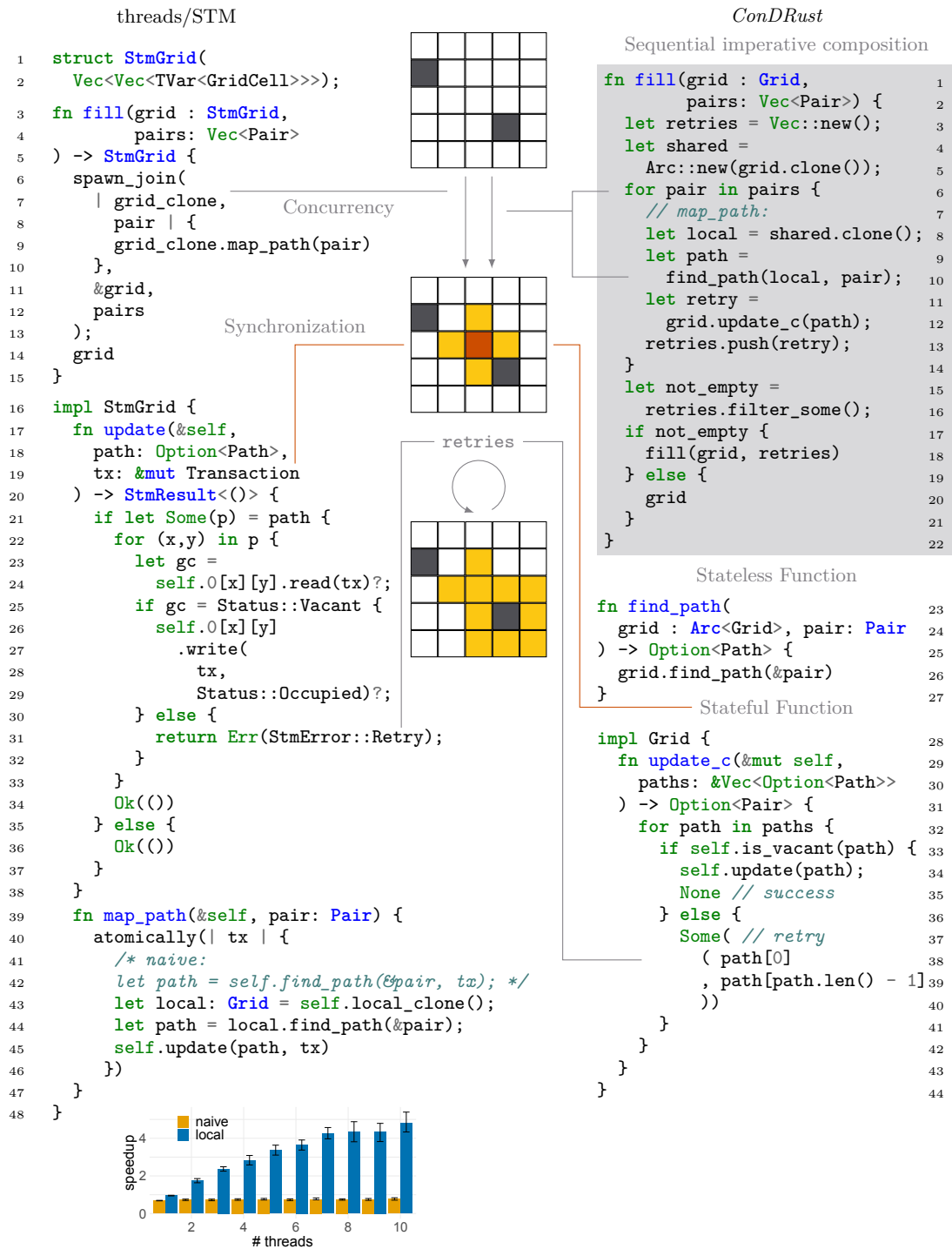
The threads/STM implementation in the left-hand column of Figure 4 starts with a re-definition of the grid data structure. Cells hold the state of the grid that the algorithm mutates and hence must be protected to prevent data races. Wrapping the cells into transactional variables (TVars) means that all grid methods need to be redefined for two reasons: First, accesses are now through the TVar. Second, each access needs a transaction which is an additional parameter to every grid method. That is the reason why all benchmarks in STAMP are implemented twice: with and without STM.

Concurrency is introduced in Line 7 of the Threads/STM implementation. We deliberately assume a higher-order function `spawn_join` to abstract from the verbose spawn-join pattern given in Figure 2. For every pair, the closure from Line 7–10 tries to map a path on the grid. The combination of the loop over the pairs and the closure’s `move` semantics for threads demand a clone of the grid for every loop iteration. The `update` method of the new `StmGrid` then tries to occupy the corresponding cells on the grid for a computed path. When a path cell is already occupied, a retry error (Line 30) aborts the transaction and triggers a retry.

For this particular example, the `map_path` method is particularly important for scalability reasons. The general structure of `map_path` is straightforward. It consists of a closure with the two steps of the algorithm (Lines 39–46): finding a path (Line 44) and updating the grid (Line 45). Line 39 places the closure onto a transaction. The key to scalability lies in the updates to the shared state. A naive implementation for `find_path`, as shown in the comment on Line 42, would operate directly on the grid (`StmGrid`) through the TVar (see Line 24). As a result, the transaction’s read set increases dramatically which leads to a much higher probability for collisions in `update`. A common pattern is thus to have the path-finding operate on a non-transactional clone of the grid (`local`). To illustrate the effect, the bar plot below the `map_path` function compares the scalability of the naive version to that of the optimized one.

The function `map_path` is representative of a common pattern found in irregular applications. The optimization described above requires the developer to find the maximal set of accesses that still preserves data-race freedom. This, in turn, requires carefully distinguishing the parts of the program that have side-effects on the state from the ones that are pure. Making this distinction explicit is a core idea behind the *ConDRust* programming model.

³ <https://github.com/Marthog/rust-stm>



■ **Figure 4** Introducing concurrency into the labyrinth benchmark adds collisions. The left column shows the implementation with threads and STM. The right column shows the imperative sequential *ConDRust* program.

3.2.2 *ConDRust*

The right-hand column in Figure 4 presents the *ConDRust* version of the labyrinth benchmark. We start with an informal introduction to the *ConDRust* programming model. We then explain how concurrency and synchronization arise naturally from a *ConDRust* program while preserving determinism and verifiability.

3.2.2.1 Programming model

A *ConDRust* program consists of three abstractions: *functions*, *stateless function calls* and *stateful function calls*.

► **Definition 1** (*ConDRust Functions*). A *ConDRust function* is a top-level function or an anonymous (lambda) function definition in the host language that the *ConDRust compiler* translates into a concurrent dataflow graph.

We highlight the *ConDRust* function for `fill` and `map_path` with a gray background. In the actual code base, the two functions would be located in a dedicated Rust module that is input to the *ConDRust* compiler. A *ConDRust* function may use control flow constructs of the host language, call other *ConDRust* functions or call stateless/stateful functions. We focus on loops because conditionals are rather unimportant when it comes to concurrency.

► **Definition 2** (*Stateless Function Call*). A *stateless function call* is a (host language) term of the form $f(t_1, \dots, t_n)$ where t_1, \dots, t_n are terms and f is a function symbol.

► **Definition 3** (*Stateful Function Call*). A *stateful function* is a (host language) term of the form $t_s.f(t_1, \dots, t_n)$ where t_s, t_1, \dots, t_n are terms and f is a function symbol.

Stateless functions such as `find_path` (Lines 23–27) represent pure computations to the *ConDRust* compiler. Stateful functions such as `update_c` (Lines 29–43) represent computations with side-effects to a particular state. The implementation of the stateless and stateful functions are outside the realm of the *ConDRust* compiler. That is, the developer may use the full set of the host language’s, i.e., Rust’s, features inside these functions without violating the stateless/stateful semantics.

The *ConDRust* programming model is general enough for embedding it into other languages such as Java or Python. But Rust is particularly well-suited because it allows to enforce state encapsulation via its type system. That is, the *ConDRust* compiler can reject programs with stateful function calls that return references to their state. Our Rust embedding does not support references as of now. Moving forward, we are interested in *ConDRust* in the context of share-nothing software architectures (e.g., serverless computing), where references do not exist. In the shared-memory context of this paper, the developer can either pass data by value or by reference using Rust’s `Arc`’s. In Section 4, we present the details of the type system for *ConDRust*.

3.2.2.2 Concurrency and synchronization

The *ConDRust* compiler translates the input program into a concurrent dataflow graph. In this graph, pipeline and task-level parallelism arise naturally from the data and control flow dependencies in the program. Our transformation, defined in Section 5, introduces data parallelism for stateless function calls inside loops. In the labyrinth benchmark, the `find_path` computation is a good candidate. It is located inside the loop of `fill` function and actually does not have side effects on the grid. But in the sequential version of Figure 3,

`find_path` is a method on the grid. To tell the *ConDRust* compiler that this computation is stateless, the developer turns the method into a stateless function (Lines 23–27). At Line 26, the stateless `find_path` function just wraps the `find_path` method on the `Grid` data type.

Note that the *ConDRust* version of `map_path` (right column, Lines 7–12) is almost identical to the STM/threads version (left column, Lines 38–46). At Line 8, the *ConDRust* compiler requires the developer to create a clone of the grid. State, in this case the `grid`, may only be used once inside a loop, i.e., the loop in `fill` at Lines 6–11. The *ConDRust* compiler forces the developer into the optimization that the STM/threads version needed to scale. As a consequence, the new `update_c` method on the `Grid` now needs to take collisions and retries into account. This is almost identical to the `StmGrid::update` method but is not based on a transaction for retrying. Instead, it returns the colliding pair at Lines 37–40.

After the loop, the remaining code in `fill` first filters the successfully mapped paths (Line 16) and then uses a (tail) recursion to retry the collisions. That is, the `fill` function explicitly defines the recursion that is implicit in the concept of a transaction. But the execution semantics is different.

3.2.2.3 Determinism

Transaction execution order in STM is non-deterministic but the generated dataflow program executes deterministically. The *ConDRust* program presented in the right-hand column of Figure 4 is a valid, i.e., well-typed, sequential Rust program. The Rust compiler translates this into a deterministically executing binary. The *ConDRust* compiler preserves the semantics of the program including the deterministic execution property. For the labyrinth benchmark, this boils down to the sequential order in which the stateful function `update_c` is called within the loop in `fill`. The *ConDRust* compiler preserves this order even though the paths are computed concurrently (in a data-parallel fashion).

In Section 5, we define a transformation to take advantage of amorphous data parallelism in irregular algorithms. This transformation produces code that limits the number of collisions, and thereby re-computations, for a single recursion round. To activate this transformation, the developer has to change the type of the input `pairs` from an ordered vector `Vec<Pair>` into a `HashSet` with deterministic iteration order. All of the transformations that we introduce in this paper preserve determinism and the semantics of the input program.

3.2.2.4 Verification

Semantic preservation allows the developer to apply and even formally verify further optimizations to the algorithm. The *ConDRust* version of the labyrinth benchmark is free of concurrency constructs. Kani fully supports `Arc`'s⁴. As such, the developer can formally verify properties of the *ConDRust* labyrinth implementation.

Zero-clone concurrent labyrinth. Note that both the threads/STM and the previously discussed *ConDRust* versions required cloning the state for scalability reasons. By cleverly using *ConDRust* and the underlying Rust semantics it is possible to avoid having to clone in the first place, while retaining determinism and verifiability. This is again a pattern that extends to other irregular applications.

The zero-clone implementation for concurrent path-finding is shown in Figure 5. For the reader's reference, the comments in the code contain the previous implementation from Figure 4. On the left-hand side, we restate the *ConDRust* version of the `fill` function from

⁴ <https://model-checking.github.io/kani/rust-feature-support.html>

```

1  fn fill(grid : Grid, pairs: Vec<Pair>) {
2      let /* retries */ paths = Vec::new();
3      let shared =
4          Arc::new(grid/* .clone() */);
5      for pair in pairs {
6          let local = shared.clone();
7          let path = find_path(local, pair);
8          /* let retry =
9              grid.update_c(path);
10             retries.push(retry); */
11         paths.push(path);
12     }
13     /* let not_empty =
14         retries.filter_some(); */
15     let (paths, grid) = unarc(paths, shared);
16     let (not_empty, retries) =
17         grid.updates(paths);
18     if not_empty {
19         fill(grid, retries)
20     } else {
21         grid
22     }
23 }

impl Grid {
1  fn updates(
2      &mut self,
3      paths: Vec<Option<Path>>
4  ) -> (bool, Vec<Pair>) {
5      let mut retries = Vec::new();
6      for path in paths {
7          let r = self.update_c(path);
8          retries.push(r)
9      }
10     let not_empty =
11         retries.filter_some();
12     (not_empty, retries)
13 }
14 }
15 }

fn unarc<S,T>(
16     s:S, t: Arc<T>
17 ) -> (S,T) {
18     match Arc::<T>::try_unwrap(t) {
19         Ok(t) => (s,t),
20         _ => panic!("Failed to unarc.")
21     }
22 }
23 }

```

■ **Figure 5** The `unarc` optimization provides a *ConDRust* version of the labyrinth benchmark that does not have to clone the grid.

Figure 4. We extracted the updates to the grid from Lines 8–9 to Lines 16–17 after the loop. This requires defining a new method (/stateful function) `updates` on the `Grid` that performs the loop over the computed paths. The corresponding code in the upper right part of Figure 5 also directly filters the retries. At this point, the key observation is that we can safely reuse the grid after the loop. At Line 4, `Arc::new` takes ownership of the grid. We had to clone the `grid` to use it inside the loop for updates and after the loop for the recursion. In the new version, the updates happen after the loop and the `find_path` actually takes ownership of the cloned `Arc` from Line 6. That is, when all path computations are done, we can safely take the `grid` out of the `Arc` again. This is what we specify at Line 15 and in the *ConDRust* version of Figure 4 at Line 24.

Figure 5 defines the stateless function `unarc` in the lower right part. The `unarc` function unpacks the `Arc`, its first argument, but leaves the second unchanged. We have to add a `panic` for the case where the `Arc` is still held elsewhere. But this is impossible by definition of Line 15. We verified this property and among the 422 reachable checks Kani reports:

```

Check 321: <std::vec::Vec<std::option::Option<benchs::Path>>
          as benchs::Unarc>::unarc.assertion.1
- Status: SUCCESS
- Description: "Failed to unarc."
- Location: src/benchs.rs:269:18
  in function <std::vec::Vec<std::option::Option<benchs::Path>>
          as benchs::Unarc>::unarc

```

Due to the semantic preserving transformations in *ConDRust*, this property also holds for the generated concurrent dataflow code.

Terms $t ::= x \mid v \mid |x : T| \rightarrow T \{ t \} \mid t(t) \mid \mathbf{let} \ x : T = t; t \mid \mathbf{let \ mut} \ x : T = t; t \mid$
 $\mathbf{f}_{\text{SL}}(t_1) \mid t_s.\mathbf{f}_{\text{SF}}(t_1) \mid \mathbf{for} \ x \ \mathbf{in} \ t \{ t \} \mid \mathbf{trfix} \ t \ t$
 Values $v ::= l \mid v_{\text{Ⓢ}} \mid |x : T| \rightarrow T \{ t \}$

■ **Figure 6** Syntactical constructs of Ⓢ_s .

Note that this `unarc` optimization is only necessary because the compiler presented in this paper does not support references. Support for references is future work but will remove such optimizations from the code and move them into the compiler. In the meantime, the developer can formally verify such optimizations with existing tools such as Kani.

4 Ⓢ_s – A subset of Rust for sequential composition

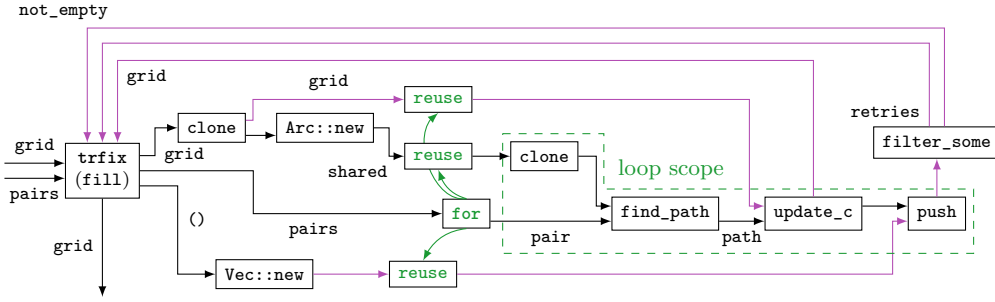
In this section, we formally specify Ⓢ_s , the subset of the Rust language that embeds the *ConDRust* programming model. Since Ⓢ_s is a subset of Rust, the operational semantics are the same as for Rust. Therefore, we restrict the presentation to the syntactical constructs. The appendix defines the type system and the operational semantics for the interested reader.

ConDRust supports the subset of Rust’s syntax that is necessary to compose calls to stateless and stateful functions (also called methods). We define this subset in Figure 6 as Ⓢ_s – a subset of Rust for sequential composition. For this paper, we restrict the terms of the language to variables x , abstractions (closures in Rust) $|x : T| \rightarrow T \{ t \}$, algorithm application $t(t)$, immutable and mutable bindings, `for`-loops and tail-recursion (`trfix`). We restrict the definition of Ⓢ_s in the following (common) ways:

1. Abstractions and calls may only have a single parameter. The extension to support multiple parameters is straightforward.
2. We desugar top-level algorithm definitions into `let`-bound closures such that a top-level defined function can be used in multiple locations of succeeding function definitions.

The key ingredient in *ConDRust*’s programming model are *stateless function calls* $\mathbf{f}_{\text{SL}}(t_1)$ and *stateful function calls* $t_s.\mathbf{f}_{\text{SF}}(t_1)$.⁵ The definition of stateless and stateful functions is not part of Ⓢ_s , as discussed before. Inside these functions, developers re-gain the full features of Rust. We further restrict control flow to loops and tail recursion leaving out other forms such as conditionals that play only a minor role in the parallel execution of a program. In Section 5 and in our implementation, loops may in fact iterate over all data types that implement Rust’s `Iterator` trait which for instance includes `HashSet`. This allows the developer to specify that the loop does not depend on a particular order and enables our second transformation that extracts amorphous data parallelism. To model this in Ⓢ_s , we assume a stateless function that uses the iterator to collect the items into a list before looping over them. In fact, `collect` is a standard function of Rust’s `Iterator`. We allow loops with an unknown iteration count via tail recursion. Tail recursion is a derived form. Precise definitions with their restrictions on variable usage are in the appendix. In the context of this paper, we are particularly interested in the case where the arguments to the recursion are a state to be updated and a worklist that triggers these updates.

⁵ We allow *ConDRust* algorithms to be called from anywhere in a Rust program. Such a call may have arguments.



■ **Figure 7** *ConDRust* dataflow graph for the (unoptimized) labyrinth code listed in Figure 4. The graph contains *state arcs* that transfer state into and out of the *loop scope*. The *for* and *reuse* nodes make sure that processing inside the loop is well-balanced, i.e., does not get stuck.

$$\begin{array}{ll}
 \downarrow_{\text{ST}} \text{ let } x = x_s.f(x_1); t & := \text{ let } (x'_s, x) = x_s.f(x_1); \downarrow_{\text{ST}} [x_s \mapsto x'_s]t \\
 \downarrow_{\text{ST}} \text{ let } x = f(x_1); t & := \text{ let } x = f(x_1); \downarrow_{\text{ST}} t \\
 \downarrow_{\text{ST}} \text{ let } _ = \text{ for } x_1 \text{ in } x_2 \{ t_3 \}; t_4 & := \text{ let } _ = \text{ for } x_1 \text{ in } x_2 \{ \downarrow_{\text{ST}} t_3 \}; \downarrow_{\text{ST}} t_4 \\
 \downarrow_{\text{ST}} |x : T| \rightarrow T \{ t \} & := |x : T| \rightarrow T \{ \downarrow_{\text{ST}} t \} \\
 \downarrow_{\text{ST}} t & := t
 \end{array}$$

$$\begin{array}{ll}
 \uparrow^{\text{STL}} \text{ let } _ = \text{ for } x_1 \text{ in } x_2 \{ \uparrow^{\text{STL}} t_3; \text{ let } (x'_s, x_3) = x_s.f(x_1); t_4 \}; \uparrow^{\text{STL}} t & := \text{ let } (x'_s) = \text{ for}^* x_1 \text{ in } x_2 \{ \uparrow^{\text{STL}} t_3; \text{ let } (x'_s, x_3) = x_s.f(x_1); t_4; (x'_s) \}; [x_s \mapsto x'_s](\uparrow^{\text{STL}} t) \\
 \uparrow^{\text{STL}} (\uparrow^{\text{STL}} t) & := (\uparrow^{\text{STL}} t)
 \end{array}$$

■ **Figure 8** Transformation of an imperative program into a functional one based on state threads.

Limitations. Currently, $\textcircled{\text{S}}$ does not include include references and in particular borrowing. The support for references and their translation into dataflow is interesting future work.

5 Compiling *ConDRust* algorithms to dataflow

In this section, we describe the main steps in *ConDRust* compilation from an imperative algorithm to a dataflow graph. The dataflow representation of the program is not the usual program dependence graph that is used in classic compilers such as LLVM for dataflow analyses. The dataflow graph that *ConDRust* targets is a runtime representation and parallel execution model of a program. Dataflow runtimes are the foundation for scalable database engines, data streaming for embedded systems and data analytics [38, 23, 27, 58]. This dataflow model is a perfect fit for such systems because it makes parallelism explicit in the graph. In this section, we focus primarily on the 3 forms: task-level, pipeline and data parallelism. We start with the translation of algorithms into dataflow and present the dataflow representation informally to define our transformations for data parallelism. In Section 6, we formally specify the semantics of the dataflow graph construction and execution as part of *ConDRust*'s code generation process.

5.1 From sequential-imperative to parallel-functional dataflow

ConDRust's programming model with its restrictions on variable usage enable the compiler to translate a $\textcircled{\text{S}}$ program into a dataflow graph that exposes pipeline and task-level parallelism while preserving the program's semantics. This translation encompasses two steps that we

define in Figure 8. In the first step, *ConDRust* translates an algorithm in \mathbb{E}_s into applicative normal (ANF) form. In \downarrow_{ST} , every call to a stateful function becomes a *state thread (ST)*: **let** $(x'_s, x) = x_s.f(x_1)$. The state x'_s is the updated state after the call. To make sure that succeeding stateful function calls on x_s operate on the new state x'_s , we substitute x_s with x'_s in t . In the second step, the compiler removes the global notion of imperative state and effectively transforms the program into a functional one. This transformation relies on the notion of state threads [35, 56, 25]. \uparrow^{STL} turns every loop into a state thread, i.e., a *state-threading loop (STL)*. The resulting states of a loop are all states of the state threads inside the loop. We restrict the definition to a single state x'_s for brevity. \downarrow_{ST} rewrites the term before recursing into the subterms. To handle nested loops, \uparrow^{STL} recurses into the subterms first and rewrites the current term with the already rewritten subterms.

From this functional program representation, the *ConDRust* compiler translates stateless and stateful calls into nodes. Data dependencies become arcs that transfer data values in FIFO order. We denote the different types of nodes in a *ConDRust* dataflow graph as follows:

$$n ::= \boxed{f_{\text{SL}}} \mid \boxed{f_{\text{SF}}} \mid \boxed{\text{for}} \mid \boxed{\text{reuse}} \mid \boxed{\text{trfix}}$$

The first two node types execute calls to stateless and stateful functions respectively. In order to perform a call, a node needs to retrieve a data value from each of its incoming arcs and emits the result of the call to its outgoing arc before the next call is constructed. Stateful nodes additionally emit their updated state via a dedicated outgoing arc.

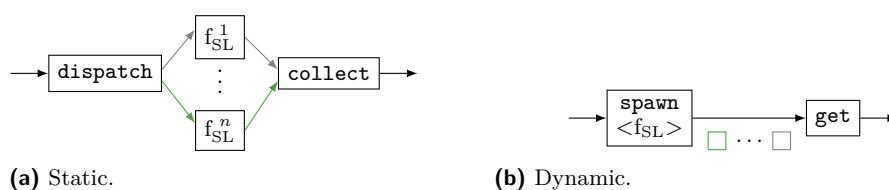
ConDRust translates loops and tail recursions directly into dedicated dataflow nodes. The **for** node streams the elements of the vector into its outgoing arc. The **trfix** node ties the knot of the recursion. Both language constructs, loops and tail recursion, open a new contextual scope, i.e., a subgraph. For tail recursion, this subgraph is closed such that the only way for data to enter and leave the graph is the **trfix** node. For loops, data enters the subgraph via **for** and **reuse** nodes and leaves it via a stateful function call node.

The dataflow construction is best explained on the dataflow graph for the labyrinth benchmark, shown in Figure 7. *ConDRust* generated this graph from the \mathbb{E}_s specification on the right of Figure 4. Data that enters the loop subgraph via the **for** node *drives* the computation. State variables entering the loop body are **retries** and **grid**. The corresponding arcs are gated by **reuse** nodes that receive the data entering the loop. The **reuse** node attaches a reuse count n where n is the number of loop iterations, i.e., elements in the vector of pairs. Function call nodes with such a reuse count as input reuse the data values for n calls.

Task-level and pipeline parallelism

Task-parallelism automatically arises whenever nodes are (data) independent of each other, such as for example the **clone** node and the **Vec::new** node. The **for** node introduces pipeline parallelism between all data dependent nodes in the loop scope subgraph. As such the computation of the path for the third pair can already start while at the same time, the grid is updated with the found path for the second pair and the result of the first computed path is pushed onto **retries**, the result vector.

Task-level and pipeline parallelism fall out naturally from a dataflow representation of a program, but they are insufficient to compete with a threads/STM-based program. This is because exploiting data parallelism is key for scalability. Almost all programs in shared memory benchmark suites such as STAMP and PARSEC contain some data-parallel part [41, 8]. Apart from data parallelism, scalability in threads/STM implementations also depends on the retry overhead introduced by the STM in case of collisions.



■ **Figure 9** Data parallelism inside a dataflow graph.

5.2 Dynamic data parallelism in a static dataflow

Data parallelism arises from an implicit (in-)dependence between the same stateless function call across loop iterations. As such, every stateless function call inside a loop is an opportunity for data parallelism. The `find_path` call in the labyrinth algorithm is an example of this. But introducing data parallelism into a static dataflow graph as shown in Figure 9a may lead to suboptimal performance. This is especially the case when the n nodes $f_{SL}^1 \dots f_{SL}^n$ feature different computation times for different input values. For example, `find_path` executes the same code but some pairs are more difficult to connect than others. As such the deterministic merge in the `collect` node stalls, waiting for straggling work [26]. This stalling does not occur in the threads/STM execution because STM does not enforce a deterministic order.

The performance problems of static work assignments are well known. This motivated Cilk’s dynamic dataflow model (fork/join) and its work-stealing runtime scheduler [11]. To mitigate these problems without sacrificing determinism, we integrate dynamic dataflow into our static dataflow graph. In a dynamic dataflow graph, nodes are created at runtime. A node executes a task, i.e., a stateless function call, that gets *spawned* (*forked*) on demand and executes once. Spawning a task creates a handle to its *future value*, i.e., the result of the stateless function call. This handle provides a `get` method to *join* the forked task with the spawning task by blocking until the call completed and the result is available. In Rust, the API for futures is equivalent to `thread spawn` and `join` as presented in the thread/STM version in Section 3. Tasks are processed by a pool of threads, as in Cilk. Whenever a thread is idling, it may steal tasks from other threads to reduce idle time. In the case of the labyrinth benchmark, a thread that already finished its path computation may steal queued path computations from a thread with a long-running path computation.

Determinism. The transformation in Figure 9b integrates dynamic dataflow to data-parallelize nodes with stateless function calls and uses the static dataflow to preserve the data value order, i.e, determinism and the semantics of the algorithm. Instead of replicating the stateless function call f_{SL} node, we lift it into a `spawn<fSL>` node. For every received input, when normally a stateless function call would be executed, the `spawn<fSL>` node submits this computation as a task to a work-stealing runtime system and emits the corresponding *future*. The downstream `get` node retrieves the value from the future. No reordering takes place because both `spawn<fSL>` and `get` are stateless function call nodes in the static dataflow graph connected via a FIFO channel.

5.3 Amorphous data parallelism

The (data) parallelism in threads/STM-based programs is implicitly affected by two aspects: the available compute cores of the system and the operating system scheduler. Both impact the number of collisions. Our amorphous data parallelism transformation makes these implicit

effects explicit in the dataflow graph and exposes a knob to fine-tune runtime performance at compile-time. We first explain how the implicit effects influence the performance of threads/STM vs. *ConDRust* programs. Then we describe our transformation.

The implicit cap in threads/STM

In the *ConDRust*-based version of the labyrinth algorithm, the number of collisions per round is capped by n , the number of input **pairs**. In the worst case, $n - 1$ paths need to be recomputed. This is independent of whether the algorithm executes sequentially or is compiled into a dataflow graph for concurrent and parallel execution. Although the STM-based version spawns the same n threads/computations, it is unlikely for this implementation to hit this upper bound. For example, when we input 256 pairs then 256 threads race for updating the grid. In a system with 24 cores, the operating system scheduler will delay the execution of most of the threads. Putting concurrency aside for easier analysis, the first “round” would consist of 24 threads, of which at most 23 would collide. We refer to this bound as *collision limit*. The limit defines how many computations will see an outdated **grid**, i.e., shared data structure, and could thus lead to collisions. In the case of the *ConDRust* version, the algorithm defines the limit to be 256, i.e., all pairs, (for the first round) which translates into the worst case collision count of 255.

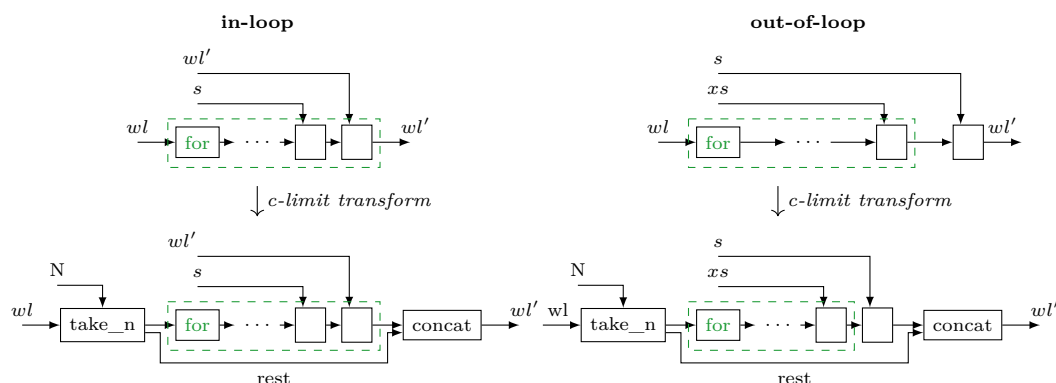
Setting a cap into irregular *ConDRust* algorithms

To compete with STM implementations, the *ConDRust* programming model makes the cap on the collision limit explicit in the algorithm. We do so by automatically transforming the algorithmic skeleton used across irregular applications to update state. As exemplified by the labyrinth benchmark, irregular applications (tail-)recurse over a worklist wl to evolve a complex data structure, i.e., a state s . In the benchmark, the worklist updates the grid.

The transformation is shown in Figure 10. To make sure it preserves the semantics, the worklist wl must be of type **Set**, i.e., the developer has to explicitly specify that there is no particular order for the elements of the worklist. The transformation distinguishes two different structures. In the *in-loop* version, the state s is updated inside the loop. The unoptimized *ConDRust* version of the labyrinth benchmark from Figure 4 is an example of this. In the *out-of-loop* version, the state update occurs at some point after the loop. This structure occurs in the **unarc** version of the *ConDRust* labyrinth implementation presented in Figure 5. In both cases, the **take_n**-node extracts the first N data items from the worklist wl and concatenates the *rest* with the recomputations *after* s was updated. Now, the compiler can optimize the parameter N , something that is not possible for threads/STM programs. N is an interesting target for further research in compiler optimization.

Determinism. Even though this transformation relies on the developer specifying that the worklist is a set, it preserves a deterministic execution. This holds as long as the set implements a deterministic iteration order when the same elements are inserted. For example, in Rust, several libraries exist that provide this property to hash set and hash map implementations.⁶ Intuitively speaking, when the worklist is a set then the algorithm is independent of a particular iteration order. Our transformation essentially picks one of these orders at compile time. But when the generated program is executed then it will always be the same deterministic order that the worklist is being processed.

⁶ <https://crates.io/crates/deterministic-hash>
https://docs.rs/hash_hasher/latest/hash_hasher/



■ **Figure 10** Transformation for amorphous data parallelism.

Terms $t ::= x \mid v \mid n \mid c; t \mid \text{run}(t, t)$
 Nodes $n ::= \text{NSL}(\text{f}_{\text{SL}}, t, t) \mid \text{NSF}(\text{f}_{\text{SF}}, t, t, t, t) \mid \text{for}(t, t, t) \mid \text{reuse}(t, t, t) \mid \text{trfix}(t, t, t, t)$
 Channels $c ::= \text{let } x = \text{chan}(t) \mid \text{let } (x, x) = \text{chan}()$
 Values $v ::= l \mid v_{\text{Ⓜ}}$

■ **Figure 11** Syntactical constructs for terms and values of Ⓜ_p .

6 Ⓜ_p — A subset of Rust for parallel composition

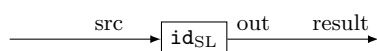
With the transformations in place, we now formally specify the backend of the *ConDRust* compiler. We present the syntactic constructs for Ⓜ_p — a subset of Rust for parallel composition, that the *ConDRust* compiler targets in Figure 11. Ⓜ_p terms basically consist of two parts:

Graph construction An arc is a channel (c) in Rust’s message-passing terminology and we define n , i.e., a term for each type of node in the dataflow graph.

Graph execution We abstract over an explicit implementation of a scheduler for a dataflow graph with a single `run` construct.

For the construction, we abstract over a concrete channel implementation. All we rely upon is the FIFO ordering property. Composition of nodes via arcs works solely via variable bindings. For example, the term in Figure 12 constructs a graph with a single (stateless) identity function (`idSL`) call node. For execution, we pass the receiving endpoint `result` and the list of nodes to `run` which executes the graph and reduces to the final result.

```
let src : Recv<i32>      = chan(5);
let (result : Recv<i32>,
    out : Send<i32>)    = chan();
run(result, (NSL(idSL, src, out) ~: [ ]))
```



■ **Figure 12** A Ⓜ_p program with a single identity node and the corresponding dataflow graph.

```
let (out, result) = std::sync::mpsc::channel();
let mut nodes = Vec::new();
nodes.push(Box::new(move || -> _ {
  let x = id(5);
  out.send(x)?;
  Ok(())
}));
run(nodes);
result.recv()?;
```

■ **Figure 13** The generated code for the graph of Figure 12.

We assume a type `Node` for nodes and align our specification for channels closely with `std::mpsc::channel` from Rust’s standard library where `Receiver<T>` and `Sender<T>` represent the receiving and the sending endpoint of a channel, respectively. In our encoding, the types `Recv<T>` and `Send<T>` are reference types (in T_{ref}) for locations l in the store μ , i.e., channels are values in the store. The types T , the context Γ (without usage tracking), the store μ , the store typing Σ and the environment Δ follow the specification in TS_s . The appendix has the complete definition of the syntax, the operational semantics of TS_p and a sketch of the proof for determinism. Informally, dataflow graphs in TS_p are essentially Kahn Process Networks (KPN) [31]. KPNs execute deterministically because incoming arcs have blocking semantics⁷ and the executed code of the node is scott-continuous. Our evaluation relation adheres to both of these properties.

7 Implementation

The current prototype of *ConDRust* comes with batteries included. No need for the developer to provide any specific implementations for channels, nodes or even a scheduler. *ConDRust* is currently implemented in 20K lines of Haskell code and takes advantage of existing language parsers with defined abstract syntax tree data structures for Rust⁸. Our implementation slightly diverges from the formal description of TS_p in terms of the `reuse` nodes. The *ConDRust* compiler implementation contains additional transformations in the backend to fuse `reuse` nodes with their downstream neighbours into a single node. That way we do not have to define reuse and non-reuse versions of all the nodes. Otherwise, our backend generates code that closely aligns with the formalization in Section 6. Figure 13 presents the generated code for the single-id-node graph of Figure 12. The *ConDRust* compiler does not create source channels but inlines values directly into the corresponding nodes. The generated Rust code uses Rust’s channels from the standard library and creates a closure for each of the nodes in the dataflow graph. The code generator moves the channels into the closure of the node such as for example `out`, the sending endpoint of the channel for the final result. The `run` function just spawns a thread for each of the nodes and rejoins them, just as in the threads/STM code of Figure 2. For the dynamic dataflow part, the compiler generates code that uses the `tokio` runtime which provides a work-stealing scheduler.⁹ *Our compiler generates safe Rust code and as such the Rust compiler verifies the absence of data races.*

Limitations. Our current implementation does not yet fully implement the type system that we formally specified in Section 4. In particular, we did not yet rigorously implement the guard for amorphous data parallelism transformation that checks whether the worklist is indeed a (hash) set. This is not due to a fundamental restriction. We believe that implementing this is straightforward and thus focused on more challenging aspects, such as implementing the transformations and code generation.

8 Evaluation

We evaluate *ConDRust* on benchmarks from 3 different benchmark suites. Our selected benchmarks cover a broad spectrum ranging from stateless to irregular algorithms. In our evaluation, we seek to answer the following questions:

⁷ Blocking semantics prevent the construction of a non-deterministic merge node, the explicit notion of non-determinism in dataflow [2].

⁸ <https://github.com/harpocrates/language-rust>

⁹ <https://github.com/tokio-rs/tokio>

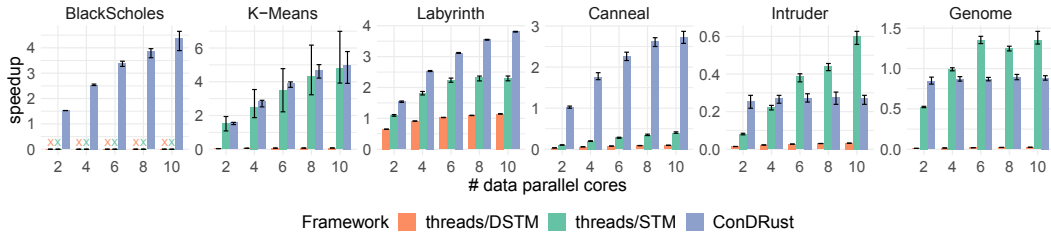
Name	Benchmark Suite	State	Algorithm	Data	I/O
			Type	Parallelism	
BlackScholes	PARSEC [8]	–	Regular	Dynamic	–
K-Means	STAMP [41]	Point-cluster assignment	Regular	Dynamic	–
Labyrinth	STAMP [41]	3D grid of coordinates	Irregular	Amorphous	–
Canneal	PARSEC [8]	Interconnected Mesh	Irregular	Amorphous	–
Intruder	STAMP [41]	Hash map	Regular	State local	–
Genome	STAMP [41]	Hash map	Regular	State local	–
Key-value store	YCSB [19]	Nested hash map	Regular	Dynamic/ State local	✓

■ **Figure 14** Benchmark description.

1. Is the *ConDRust* programming model expressive enough for a broad range of applications?
2. What is the effort to translate the sequential benchmark code to *ConDRust* code? (Appendix Section A.2 has a first comparison of the effort to convert sequential programs into threads/STM programs and *ConDRust* programs.)
3. Can the *deterministic* code, that the *ConDRust* compiler generates, deliver performance that is on par with the *non-deterministic* threads/STM-based code?
4. What is the effect of the amorphous data parallelism transformation from Section 5.3?

Benchmarks. Figure 14 summarizes the benchmarks used in our evaluation. We selected 7 benchmarks from 3 different benchmark suites: STAMP [41], PARSEC [8], and YCSB [19]. STAMP is a benchmark suite intended to investigate the performance of software and hardware transactional memory implementations. Benchmarks in STAMP basically fall into two categories: (1) Algorithms where most of the execution time is spent inside transactions and (2) algorithms with very small transactions. Transaction size directly correlates to the amount of computation that depends on the global state structure. We selected several benchmarks from both categories. Labyrinth, Intruder and Genome fall into the first category, K-Means is located in the second. K-Means clustering spends only 7% of the execution time inside transactions. No other STAMP benchmark spends fewer cycles inside transactions. Labyrinth is one of the 3 benchmarks in STAMP that spend nearly 100% of their execution time inside lengthy transactions. It is also one of 2 STAMP benchmarks with an irregular algorithmic structure. The other benchmarks in STAMP have similar characteristics to the ones that we selected [41]. SSCA2 has characteristics similar to K-Means while Yada (Delaunay Mesh Refinement) and Bayes are similar to Labyrinth. Vacation is similar to Genome. Vacation simulates database transactions with STM which is not possible in real-world database systems where transactions involve network I/O between the client and the database server. For a more realistic setting with I/O, we chose YCSB, the state-of-the-art benchmark for key-value stores. We selected 2 more benchmarks, Canneal and BlackScholes, from PARSEC that both fall into the second category with small transactions. Canneal performs simulated annealing and is also irregular. Transactions in Canneal are short but the overall time spent inside transactions is about 70%. BlackScholes is the baseline for linear scalability. Overall our benchmark set consists of ca. 12k lines of Rust code.

Setup. Our experiments ran on an Intel Core i9-10900K CPU with 3.70 GHz, 32 GB RAM and 20 hardware threads, i.e., 10 cores and 10 hyperthreads. The operating system was Ubuntu version 20.04. We used the latest `rust-stm` version 0.4.0 and extended it to support deterministic transactions [49]. We executed each experiment 30 times and report the mean.



■ **Figure 15** Speedup comparison of threads/DSTM, threads/STM and *ConDRust* across different benchmarks. Baseline is the sequential version.

Whenever possible, we used the data sets of the original benchmarks. Otherwise, we ported the data generation too. Appendix Section A has the input configurations.

Metrics. For our experiments, we ported the C/C++ reference implementations from STAMP and PARSEC to safe Rust. For each benchmark, we created 4 versions:

sequential a sequential baseline implementation,

threads/DSTM a concurrent version based on threads and DSTM, and

threads/STM a concurrent version based on threads and STM,

ConDRust a *ConDRust* version.

Benchmarks in STAMP, PARSEC and other benchmark suites (such as Lonestar for Galois programs [32]) for concurrent programming do not address the problem of finding the best granularity of work to place onto a thread. The benchmark code explicitly splits work into chunks and the size of these chunks is determined by the number of parallel threads. We follow this principle because *ConDRust* does not address the thread granularity problem either. For the STAMP and PARSEC benchmarks, we report the speedup over the sequential baseline. For the YCSB benchmark, we measure throughput.

Since we are particularly interested in the exploitation of data parallelism, we vary the *number of data parallel cores*. This is the natural metric for these applications and their respective threads/STM implementations. For the *ConDRust* versions, there are more threads because every dataflow node is assigned its own thread and we vary the number available threads for the dynamic part of the dataflow graph. Although *ConDRust* executes deterministically, we expect *ConDRust* programs to have performance that is on par with the threads/STM version if the collision-limit is tuned properly. For this reason, we explore different values of the limit and compare with the threads/STM performance. Auto-tuning approaches or heuristics can be used in the future to automatically tune the collision-limit.

8.1 Benchmark study

Figure 15 shows the overview of our benchmark study. For BlackScholes, K-means clustering, Labyrinth, and Canneal, the deterministic *ConDRust* programs outperform even the non-deterministic threads/STM counterparts. Benchmarks for genome sequencing and intrusion detection exploit data parallelism that *ConDRust* cannot yet exploit. In all of these benchmarks, DSTM synchronization delivers poor performance. In the following, we analyze the benchmark results from left to right and present further details. We add features one at a time. We start with dynamic data parallelism, then investigate irregular algorithms and then turn to *ConDRust*'s limitations. Afterwards, we case study a key-value store.

Dynamic data parallelism

The first benchmark is BlackScholes, a bulk data parallel workload. We present two versions. The first version is most intuitive. It creates vectors for the results on-demand and joins these vectors into a single result vector. That is, memory allocation is interspersed with the computation of stock options that does not require a shared data structure. This final result vector is shared state that requires synchronization. The second version pre-allocates all memory. This is the version that is most commonly used in the literature [16].

In the experiment of Figure 15, we used the intuitive benchmark version and naively opted for DSTM and STM to do the synchronization. The single-threaded *ConDRust* version completed the calculation of 40M stock options in 650 ms. The DSTM/STM versions timed out after 10 minutes. This shows that protecting a data structure blindly with STM is not really a practical solution. Efficient transactions have to be fine-grained and as such require a re-implementation of the data structure. That is why STAMP includes several dedicated STM-based data structure implementations.

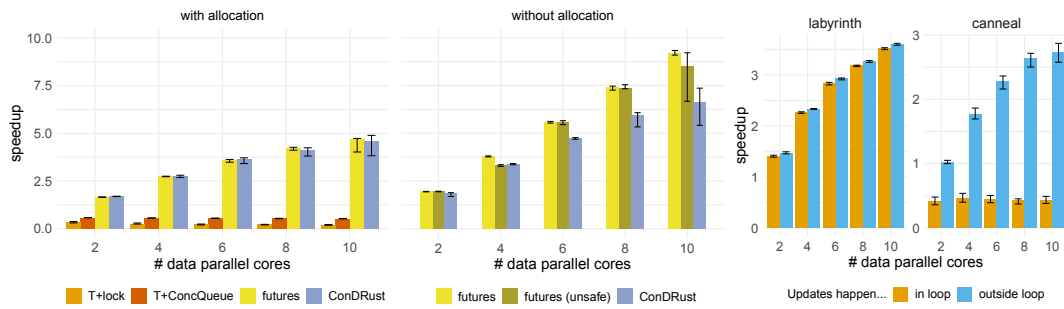
To be fair, we also compared the *ConDRust* version against 3 more versions. The first version (T+lock) uses a lock instead of a transaction to protect the data structure. Figure 16 shows that this version does finish but features poor performance. Protecting large data structures with locks is no option either, and thus fine-grained locking is required. To explore this, we use an available implementation of a concurrent queue. But even this T+ConcQueue version does not scale well either. Finally, in Rust, threads are implemented as futures, i.e., when joined, they return a result of the computation on the thread. Figure 16 shows that only futures deliver performance comparable with the dataflows that *ConDRust* generates.

We also evaluated the second version of the benchmark that pre-allocates the memory. The *ConDRust* and futures versions both pre-allocate the individual result vectors for the parallel computations. To avoid the memory-allocations and copies to produce a single flat result vector, these version return a vector with the nested individual result vectors, i.e., a vector of vectors. We also created a futures (unsafe) version that mimics exactly the C version of PARSEC. That is, it pre-allocates the result vector and passes ranges to the individual computations where the results can be written to. In safe Rust, we did not find a way to tell the borrow checker that these ranges do not overlap and data races cannot occur. Hence, we had to introduce unsafe code. The results in Figure 17 show that the *ConDRust* version benefits from pre-allocating memory but is not yet on par with the futures versions. This is due to the fact that the computations are rather small such that additional runtime overhead of the concurrent *ConDRust* code becomes visible. We are certain that optimizations are possible that reduce this runtime overhead further.

K-means clustering is the first recursive algorithm with state. The K-means plot in Figure 15 shows that scalability of deterministic *ConDRust* programs is on par with the non-deterministic threads/STM counterparts.

Amorphous data parallelism

Labyrinth and Canneal are irregular applications. The plots in Figure 15 show that *ConDRust* generates programs that even outperform the threads/STM versions. This is because costly synchronization overhead is not present in the generated dataflow programs. In both benchmarks, we used the `unarc` optimization from Section 3 to avoid state cloning. *ConDRust* performed both transformations from Section 5, for dynamic data parallelism and amorphous data parallelism. To gain insights into the algorithm structure and the effects of these transformations, we perform further analyses.



■ **Figure 16** BlackScholes with memory allocation.

■ **Figure 17** BlackScholes without memory allocation.

■ **Figure 18** In-loop vs. out-of-loop state updates.

State update placement. We study the effect of placing the update to the state inside or outside the loop that iterates the worklist (Section 5.3), with results shown in Figure 18. The in-loop version has the update to the state inside the loop. Respectively, the generated dataflow combines data parallel computations with pipeline parallel state update. The outside-loop version performs the state update only after the loop when all computations have been computed. Pipeline parallelism does not arise but the algorithm can leverage the `unarc` optimization from Section 3 for a zero-clone version. The results show that pipeline parallelism does not really lead to speedups on either of the two benchmarks. In the case of labyrinth, both versions have the same performance and for canneal, the in-loop version does not scale at all. Pipeline parallelism only pays off when pipeline stages are balanced. This is not the case for both benchmarks where the first stage computes while the second stage only updates. The results also show that the performance of the Labyrinth benchmark is not sensitive to the update placement. This is mainly because of the low overhead incurred in cloning the state of the labyrinth. Canneal has a much larger state which explains the bad scalability of the in-loop update.

Collision-limit. To study the effect of the amorphous data parallelism transformation in *ConDRust*, we compiled both benchmarks once with and once without this transformation. Figure 19 shows that the amorphous data parallelism has no effect on the performance of Canneal but has a big impact on the performance of Labyrinth. The plots in Figure 20 vary the collision-limit (*c-limit*) as a multiple of the thread count such that work distributes evenly across the data parallel workers (i.e., threads). The right-most bars in the plots show the performance without amorphous data parallelism. In Canneal (re-)computation is cheap and the state large. Setting a low collision limit just prevents data parallelism to take full effect. In Labyrinth, finding a path is expensive and so are re-computations. As such, the collision-limit for optimal performance is only a small multiple of the thread count. With our optimization, the compiler can tune performance along these complexity coordinates: state size and (re-)computation complexity.

Threads/STM. For the Canneal benchmark, the *ConDRust* version in Figure 15 even greatly outperforms the threads/STM version. In fact, the threads/STM version for Canneal does not scale at all. This is due to the characteristics of the algorithm. The workload issues tens to hundreds of thousands of rather *short* transactions which makes the STM overheads a dominating factor. This is not the case in the Labyrinth benchmark where long-running transactions outweigh their overhead.

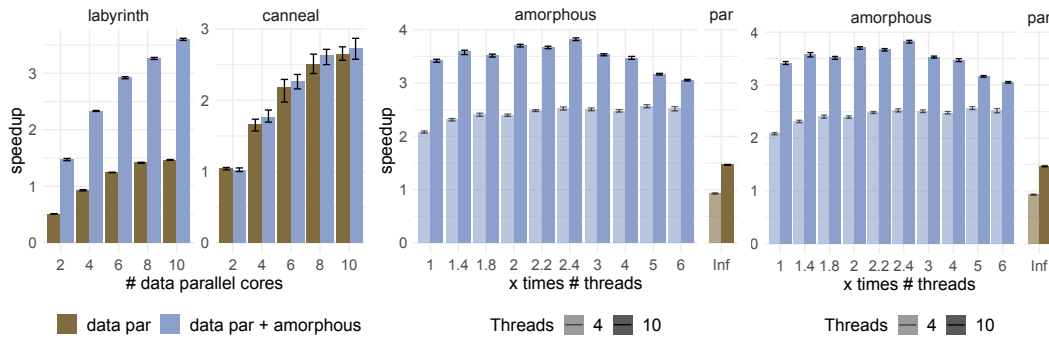


Figure 19 Amorphous.

Figure 20 Collision-limit effects.

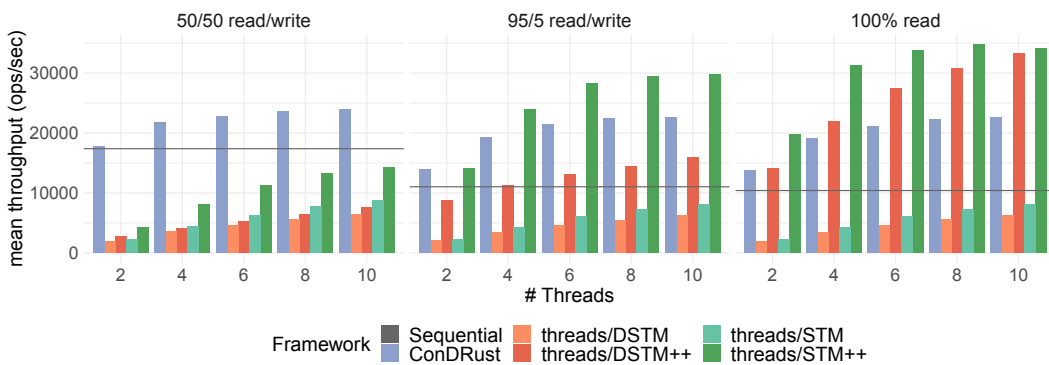


Figure 21 Throughput comparison for the key-value store implementations.

8.2 Beyond data parallelism

For Intruder and Genome, *ConDRust* fails to extract the data parallelism. Both benchmarks operate on a partitionable state structure. Intruder uses a hash map to reassemble network packets. Similarly, Genome uses a hash map to assemble and deduplicate genome sequences. The computation is stateful but operates only on a local part of the structure, for example, a bucket in the hash map. Deriving parallelism from extended knowledge about the state type and its structure is an interesting future research direction.

Conclusions. Overall, from the results in Figure 15 and our detailed analysis in Figures 16–20, we conclude that *ConDRust* generates dataflow programs that scale for certain application classes. The generated code executes deterministically while the sequential input programs remain verifiable. We managed to implement several benchmarks which establishes confidence that the programming model is expressive for a broad range of applications. Stateful functions with local effects on the state, as well as platforms with heterogeneous hardware [45] are an interesting future research direction.

8.3 Case study: Key-value store

For the YCSB benchmark, we populated the key-value store with 10,000 entries and configured a load of 30,000 operations, executed by 8 threads in parallel. We ran 3 configurations: (1) a write-heavy configuration with 50% reads and 50% writes, (2) a cloud-typical configuration

with mostly reads (95%) and (3) a read-only configuration. Figure 21 presents the throughput results for the key-value store implementations. In addition to the threads/(D)STM versions, we add another version called threads/(D)STM++. The threads/(D)STM++ version uses additional atomic-read instructions for read operations. We provide this implementation for fairness reasons because the threads/(D)STM version does not scale at all. This is due to the fact that the benchmark essentially just queries the data structure or updates it, with no substantial computation. Hence, the load creates a lot of accesses to the data structure, similar to Canneal. Even worse, STM clones the read value to provide an isolated private view on the read data. In the case of the key-value store, that data is essentially the entire underlying hash map. The atomic-read operations prevent this effect.

Even compared against the threads/(D)STM++ optimized version, the *ConDRust* generated code scales better in the write-heavy configuration and is almost on par with threads/STM++ in the cloud-typical configuration. Naturally, threads/(D)STM++ performs better when there are only reads. The *ConDRust* key-value store implementation cannot extract the parallelism from partitioning the hash map of the key-value store, similar to the Intruder and Genome benchmarks. The takeaway of this experiment is that the developer can write simple sequential code and the *ConDRust* compiler provides speedups that are on par with fine-tuned threads/STM implementations, while preserving determinism.

9 Related work

Various approaches exist to make concurrent programming deterministic but they either are not expressive enough or target functional rather than imperative programs. Language extensions such as the effect (type) system proposed by the Deterministic Parallel Java (DPJ) project primarily focus on proving the deterministic guarantees and providing these to the developer in the most non invasive way [13]. The DPJ authors conclude: “[...] *studying a wide range of realistic parallel algorithms has shown us that some significantly more powerful capabilities are needed for such algorithms.*” NESL is a functional language with the well-known higher-order functions `map` and `reduce` to parallelize stateless applications [9]. MapReduce is the programming model that has seen popularity for the very same reasons but has fallen from grace due to its limited expressivity, i.e., no states, no variables, no loops etc. None of the approaches derives scalable concurrency straight from imperative sequential programs that can be formally verified. The closest in spirit is MOLD, a tool that translates sequential imperative Java programs into MapReduce programs [46]. But MOLD does neither define a precise subset that it can translate nor reasons about verifiability or determinism of the compiled program.

Deterministic parallelism is a well-studied area but so far no approach could provide on-par performance with non-deterministic executions. To provide deterministic parallelism in MapReduce, the developer has to make sure that the function passed as an argument to `reduce` is associative and commutative. Commutativity also plays a key role in revisions, an extension to NESL’s programming model to support shared state [10]. In this case, the developer has to provide a commutative function that is used at runtime to acquire a lock on a data value. Programming-wise, this shares similarities to programming lattice-based data structures [33]. Semantic-wise, the execution of revisions is the same as for software transactions which underpin most of the runtime approaches for deterministic parallelism in one form or the other. Notable examples include deterministic Galois [42], DeSTM [49] and LiTM [57]. Even CoreDet, the only fully compiler-based approach, uses the notion of a transaction and a barrier to synchronize threads and enforce a deterministic commit

order [7]. DMP, CoreDet’s predecessor, fully relied on software transactions which did not scale because the transactions became too large [22]. CoreDet relied on hardware transactions, but implementations of these turned out to support only very small transactions and were even disabled again from Intel processors because their inherent complexity allowed various side-channel attacks [24, 18, 34]. *ConDRust* requires no additional commutativity properties nor specialized hardware. Nevertheless, we do acknowledge that performance could certainly benefit from properties such as associativity and commutativity.

We are not the first to recognize the benefits of dynamic task scheduling and the collision limit for irregular applications. However, we are the first to build atop a dataflow runtime and are not aware of a compiler with explicit transformations for these key performance concepts. Higher-level abstractions for data parallelism often build upon dynamic dataflow constructs such as Cilk’s fork/join primitives [11]. Examples include Galois collections and the parallel loops in the style of NESL in the revisions programming framework. All other approaches, use threads to let the operating system schedule operations. Similar to our collision limit, the authors of revisions perform rounds of computations in batches to bound the number of computations per round [10]. LiTM implements revisions as simple transactions and the internal algorithm that executes these transactions is almost identical to the result of the in-loop state update transformation to limit the collisions per round in Figure 10 [57]. But LiTM again inherits all the overhead that is connected with an STM implementation such as maintaining read/write sets and lock tables. *ConDRust* does not incur such overheads because there are no data races in the generated dataflow programs and as such no synchronization is required.

10 Conclusion and Future Work

We presented *ConDRust*, a new programming model and compiler to translate verifiable sequential imperative Rust programs into scalable concurrent ones. The developer can use existing tools such as Kani to formally verify the sequential program. For scalable concurrency, the *ConDRust* compiler translates the sequential composition into a concurrent one based on threads and message-passing channels. Our compiler design fosters semantic-preserving transformations that preserve interesting properties such as determinism. In our evaluation, the *ConDRust* compiler generated code that even outperformed non-deterministic concurrent programs. Our compiler is aware of stateful calls and serializes them without costly synchronization. This benefit is big enough to outweigh the cost of enforcing a particular deterministic order even for stateful irregular applications that are notoriously hard to parallelize. Our results motivate the following interesting directions for future work:

Semantic preservation In this paper, we argued only informally that our compiler transformations preserve the semantics of the input program. Nevertheless, the described transformations can serve as the foundation for a formally-verified version of our compiler.

References \oplus_s , the subset for sequential imperative composition, presented in this paper, does not include references. The developer has to use runtime-checked reference implementations (**Arcs**) and according optimizations such as the **unarc** optimization from Section 3. Adding references to \oplus_s is certainly an interesting future research direction.

Partitioned state A limitation of our programming model so far is the missing notion of functions that operate on disjoint parts of a state structure. Performance for such algorithms is not on par with their concurrent counterparts. What is a sufficient encoding of partitioned state in \oplus_s ?

References

- 1 Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space-efficient parallel functional programming. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434299.
- 2 Arvind, Kim P. Gostelow, and Wil Plouffe. Indeterminacy, monitors, and dataflow. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles, SOSP '77*, pages 159–169, New York, NY, USA, 1977. Association for Computing Machinery. doi:10.1145/800214.806559.
- 3 Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360573.
- 4 Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017. doi:10.1145/3015146.
- 5 Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, 1991.
- 6 Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20–28, 2010. Proceedings 19*, pages 283–303. Springer, 2010.
- 7 Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 53–64, 2010.
- 8 Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, volume 2011, page 37, 2009.
- 9 Guy E. Blelloch. *NESL: a nested data parallel language*. Carnegie Mellon Univ., 1992.
- 10 Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 181–192, 2012.
- 11 Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95*, pages 207–216, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/209936.209958.
- 12 Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar'09*, page 4, USA, 2009. USENIX Association.
- 13 Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, 2009.
- 14 Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In *Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4–8, 2005. Proceedings 8*, pages 395–409. Springer, 2005.
- 15 Cristiano Calcagno, Hongseok Yang, and Peter W. O’hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science: 21st Conference Bangalore, India, December 13–15, 2001 Proceedings 21*, pages 108–119. Springer, 2001.

- 16 Dimitrios Chasapis, Marc Casas, Miquel Moretó, Raul Vidal, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Parsecs: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4), December 2015. doi:10.1145/2829952.
- 17 Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–36, 2003.
- 18 Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, et al. Evaluation of amd’s advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40, 2010.
- 19 Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
- 20 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.
- 21 Enrico Armenio Deiana, Brian Suchy, Michael Wilkins, Brian Homerding, Tommy McMichen, Katarzyna Dunajewski, Peter Dinda, Nikos Hardavellas, and Simone Campanoni. Program state element characterization. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2023, pages 199–211, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3579990.3580011.
- 22 Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 85–96, 2009.
- 23 David J. DeWitt, Robert Gerber, Goetz Graefe, Michael Heytens, Krishna Kumar, and Murali Muralikrishna. Gamma-a high performance dataflow database machine. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1986.
- 24 Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14, 2014.
- 25 Sebastian Ertel, Justus Adam, Norman A Rink, Andrés Goens, and Jeronimo Castrillon. Stelang: State thread composition as a foundation for monadic dataflow parallelism. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pages 146–161, 2019.
- 26 Kim P. Gostelow and Wil Plouffe. Indeterminacy, monitors, and dataflow. In *Proceedings of the sixth ACM symposium on Operating systems principles*, pages 159–169, 1977.
- 27 Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. *ACM SIGMOD Record*, 19(2):102–111, 1990.
- 28 Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.
- 29 Wesley M. Johnston, JR. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.
- 30 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.
- 31 Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

- 32 Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76. IEEE, 2009.
- 33 Lindsey Kuper and Ryan R. Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84, 2013.
- 34 Michael Larabel. Intel to disable tsx by default on more cpus with new microcode. https://www.phoronix.com/scan.php?page=news_item&px=Intel-TSX-Off-New-Microcode, 2021. [Online; accessed 02-March-2022].
- 35 John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94*, pages 24–35, New York, NY, USA, 1994. ACM. doi:10.1145/178243.178246.
- 36 Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions on electronic computers*, pages 346–365, 1961.
- 37 E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. doi:10.1109/MC.2006.180.
- 38 Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- 39 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009. doi:10.1145/1538788.1538814.
- 40 Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, September 2020. doi:10.14778/3407790.3407808.
- 41 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46. IEEE, 2008.
- 42 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: On-demand, portable and parameterless. *ACM SIGPLAN Notices*, 49(4):499–512, 2014.
- 43 Peter O’Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.
- 44 Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *ESOP*, volume 6602, pages 439–458. Springer, 2011.
- 45 Christian Pilato, Stanislav Bohm, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, Vojtech Cima, Radim Cmar, Dionysios Diamantopoulos, Fabrizio Ferrandi, Jan Martinovic, et al. Everest: A design environment for extreme-scale big data analytics on heterogeneous platforms. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1320–1325. IEEE, 2021.
- 46 Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '14*, pages 909–927, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660193.2660228.
- 47 Mike Rainey, Ryan R. Newton, Kyle Hale, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut A. Acar. Task parallel assembly language for uncompromising parallelism. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021*, pages 1064–1079, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3460969.
- 48 Ganesan Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.
- 49 Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. Destm: Harnessing determinism in stms for application development. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, PACT '14*, pages 213–224, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2628071.2628094.

- 50 Federico Reghenzani, Giuseppe Massari, and William Fornaciari. Timing predictability in high-performance computing with probabilistic real-time. *IEEE Access*, 8:208566–208582, 2020.
- 51 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.
- 52 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- 53 Aaron Turon. *Understanding and expressing scalable concurrency*. PhD thesis, Northeastern University, 2013.
- 54 Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '22*, pages 321–330, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510457.3513031.
- 55 Philip Wadler. Linear types can change the world! In *Programming concepts and methods*, volume 3, page 5. Citeseer, 1990.
- 56 Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.
- 57 Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: A lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.
- 58 Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, et al. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

A Evaluation

This section provides the details for our evaluation. We list configuration parameters and afterwards show code metrics to compare the threads/STM programs with the *ConDRust* programs.

A.1 Configurations

Benchmark	Arguments
BlackScholes	in_40M.txt
K-Means	-n 40 -t 0.00001 random-n65536-d32-c16.txt
Labyrinth	random-x512-y512-z7-n512.txt
Canneal	-swaps 15000 -t 2000 -m 128 400000.nets
Intruder	-a 10 -l 16 -n 4096 -s 1
Genome	-g 16384 -s 64 -n 16777216
YCSB	kv-store size = 10,000 records, operation count: 30,000

■ **Figure 22** Benchmark parameters and inputs.

Figure 22 lists the configurations that we used in our experiments. Whenever possible, we used the data sets from the original benchmarks. When this was not possible, we ported the data generation too.

A.2 Programmability Comparison

Table 1 compares *ConDRust* and threads/STM in terms of the programming effort required to derive the respective implementation from a sequential one. Of course effort in itself is hard to measure, as different abstractions and frameworks require different thought processes

■ **Table 1** Comparison of the programming effort required to adapt a sequential program to *ConDRust* (ConD_{rs}) and threads/STM.

Benchmark	State		Function		Synchronization		Concurrency	
	Modifications		Recompositions		Primitives		Code	
	ConD _{rs}	STM	ConD _{rs}	STM	ConD _{rs}	STM	ConD _{rs}	STM
K-Means	0	2	7	7	0	11	0	2
Labyrinth	1	1	10	6	0	4	0	2
Canneal	12	3	12	8	0	31	0	2
Intruder	0	1	4	4	0	12	0	2
Genome	3	8	11	5	0	15	0	4

when used. Therefore, the table compares how a number of key properties of the applications in question changed. State modifications denote changes to fields of the program state and are a direct result of adapting an application to another framework. In order to derive a *concurrent composition* and accomodate state modifications, functions must be changed. These changes are denoted as function recompositions. Furthermore, the derivation of a concurrent application requires in case of the threads/STM approach the introduction of concurrency and synchronization code. Of course, some modifications prompt further changes affecting categories. A state change may require adjusting multiple function signatures and bodies, while added concurrency requires synchronization. Hence, fewer modifications are always better, as they require less effort. Note that this comparison is potentially biased, as the original STAMP suite did not include sequential versions. We derived these manually from the parallel code, which may result in the sequential versions being easier to port to the threads/STM framework.

The first difference is that *ConDRust* programs are free of concurrency abstractions and synchronization primitives. This does not only enable verification but prevents the introduction of concurrency hazards such as data races or deadlocks. We observe that increased use of synchronization primitives results in more transaction conflicts and degraded performance. As synchronization in threads/STM works on the type level, the framework requires generally more state modifications, but fewer individual function recompositions. This means that while a smaller percentage of the code base is changed, the changes are more substantial. A single recomposition here may include incorporating synchronization or concurrency, such that functions or parts thereof can be run in a transaction. Since transactions may fail, failure models have to be considered while altering the code. Also, since transactions can not be nested, special care must be taken to avoid that. Finally, transaction size plays an important role in the overall performance and must therefore be carefully chosen.

ConDRust on the other hand for the most part only needs few state changes. These are mostly done to remove types that are not thread-safe and could hence not be used in a concurrent environment. In the case of Canneal, a large struct had to be used to replace a non type safe state sharing approach. The main work required to derive a *ConDRust* implementation indeed lies in the decomposition and recomposition of functions. In contrast to threads/STM this often only entails breaking up bigger functions into several smaller ones (which are counted individually) and removing references from function definitions. As a result, the code bases became more fine-grained and compartmented, with each function only handling a single task.

Terms	$t ::= x \mid v \mid x : T \rightarrow T \{ t \} \mid t(t) \mid \text{let } x : T = t; t \mid \text{let mut } x : T = t; t \mid$ $f_{\text{SL}}(t_1) \mid t_s.f_{\text{SF}}(t_1) \mid \text{for } x \text{ in } t \{ t \} \mid \text{trfix } t t$
Evaluation context	$E ::= \square \mid E(t) \mid v(E) \mid \text{let } x : T = E; t \mid \text{let mut } x : T = E; t \mid$ $f_{\text{SL}}(E) \mid E.f_{\text{SF}}(t) \mid v_s.f_{\text{SF}}(E) \mid$ $\mid \text{for } x \text{ in } E \{ t \} \mid \text{for } x \text{ in } v \{ E \} \mid \text{trfix } E t \mid \text{trfix } v E$
Values	$v ::= l \mid v_{\text{Ⓢ}} \mid x : T \rightarrow T \{ t \}$
Types	$T ::= \text{Ref}\langle T \rangle \mid T_{\text{Ⓢ}} \mid \text{mut } T_{\text{Ⓢ}} \quad \frac{t \mid \mu \rightarrow t' \mid \mu'}{E[t] \mid \mu \rightarrow E[t'] \mid \mu'} \text{E-C}_{\text{TXT}}$
Typing context	$\Gamma ::= \emptyset \mid x : (n, T)$
Store	$\mu ::= \emptyset \mid \mu, l = v$
Store typing	$\Sigma ::= \emptyset \mid \Sigma, l : T$
Environment	$\Delta ::= \emptyset \mid f_{\text{SL}} : T \rightarrow T \mid f_{\text{SF}} : \text{mut } T \rightarrow T \rightarrow T$

■ **Figure 23** Syntactical constructs and evaluation context of Ⓢ_s .

Overall, we observe that *ConDRust* requires less severe changes to the code base. The changes that are required are merely the breaking up of functions to expose parallelism and the removal of state sharing.

B Ⓢ_s – A subset of Rust for sequential composition

In this section, we formally specify Ⓢ_s , the subset of the Rust language that encompasses the *ConDRust* programming model. We start with the syntax and the operational semantics with focus on the integration of stateless and stateful function calls. Afterwards, we specify the type system that guards the usage of state. A clear specification of state is important for the compiler to reason about the various forms of parallelism in the derived dataflow representation of the program.

B.1 Syntax

ConDRust supports the subset of Rust’s syntax that is necessary to compose calls to stateless and stateful functions (also called methods). We define this subset in Figure 23 as Ⓢ_s – a subset of Rust for sequential composition. The semantics of Ⓢ_s are the same as for Rust. For this paper, we restrict the terms of the language to variables x , abstractions (closures in Rust) $|x : T| \rightarrow T \{ t \}$, algorithm application $t(t)$, immutable and mutable bindings, **for**-loops and tail-recursion (**trfix**). We restrict the presentation of Ⓢ_s in the following (common) ways:

1. Abstractions and calls may only have a single parameter. The extension to support multiple parameter is straightforward.
2. We desugar top-level algorithm definitions into **let**-bound closures such that a top-level defined function can be used in multiple locations of succeeding function definitions.

The evaluation context E specifies that terms evaluate from left to right in a call-by-value fashion. Ⓢ_s and Rust are imperative languages such that require a store μ to model the state of the program. Store locations l are part of the syntactical constructs. The small-step operational semantics $t \mid \mu \rightarrow t' \mid \mu'$ relates a term t and a store μ to a term t' and a store μ' . The store μ maps labels to values where $\mu, l \mapsto v$ denotes the usual conjunction of store mappings μ and the mapping from label l to value v . Values are store locations, (tail recursion) abstractions and the values defined in the Rust language itself. In the specification of Ⓢ_s ’s operational semantics, we assume general types and values for booleans, tuples and lists with constructors $[]$ for the empty list and $v \sim: vs$ (**cons**) where v is the head with the tail vs . In Rust, the corresponding data structure to a list is a vector (**Vec**).

B.2 Operational Semantics

The usual way for values to enter the evaluation is via the key ingredient in *ConDRust*'s programming model: *stateless function calls* $f_{\text{SL}}(t_1)$ and *stateful function calls* $t_s.f_{\text{SF}}(t_1)$.¹⁰ The definition of stateless and stateful functions themselves are not part of \mathfrak{S}_s . We define Δ as an environment in the typing relation that holds the typing information for the stateless and stateful functions used in the term to be evaluated. As such, the operational semantics for calls to stateless and stateful functions rely upon the evaluation relation of Rust ($\Downarrow_{\mathfrak{S}}$):

$$\boxed{t \mid \mu \longrightarrow t' \mid \mu'}$$

$$\frac{f_{\text{SL}}(v) \mid \emptyset \Downarrow_{\mathfrak{S}} v_r \mid \emptyset}{f_{\text{SL}}(v) \mid \mu \longrightarrow v_r \mid \mu} \text{E-FSL} \qquad \frac{l_s.f_{\text{SF}}(v_1) \mid l_s \mapsto v_s \Downarrow_{\mathfrak{S}} v_r \mid l_s \mapsto v'_s}{l_s.f_{\text{SF}}(v_1) \mid \mu, l_s \mapsto v_s \longrightarrow v_r \mid \mu, l_s \mapsto v'_s} \text{E-FSF}$$

Inside these functions, developers re-gain the full feature set of Rust. Stateless calls do not have side-effects. Side-effects for stateful calls are restricted to a particular state location l_s in the store μ . These are the only rules that leverage Rust's evaluation relation ($\Downarrow_{\mathfrak{S}}$). In fact, calls are the only places where computation takes place while the rest of the language is for composition. Our operational semantics are based on the standard beta-reduction such that $[x \mapsto t_1]t_2$ with $x \in FV(t_2)$ replaces all occurrences of the free variable x in t_2 with t_1 . The usual rule then covers application of simple abstractions:

$$[x : T \mapsto T \{ t_2 \}(v_1) \mid \mu \longrightarrow [x \mapsto v_1]t_2 \mid \mu \text{ (E-ABSAPP)}$$

Stateless bindings solely rely on beta-reduction. Mutable bindings register values in the store.

$$\begin{aligned} \mathbf{let} \ x : T = v_1; \ t_2 \mid \mu &\longrightarrow [x \mapsto v_1]t_2 \mid \mu && \text{(E-LET)} \\ \mathbf{let mut} \ x : T = v_1; \ t_2 \mid \mu &\longrightarrow [x \mapsto l_1]t_2 \mid \mu, l_1 = v_1 \quad \text{where } l_1 \notin \text{dom}(\mu) && \text{(E-LETMUT)} \end{aligned}$$

As such, the only values in the store refer to mutable state references. We further restrict control flow to loops and tail recursion leaving out other forms such as conditionals that play only a minor role in the parallel execution of a program. Loops iterate over a list of values.

$$\begin{aligned} \mathbf{for} \ x \ \mathbf{in} \ [] \ \{ t_2 \} \mid \mu &\longrightarrow () \mid \mu && \text{(E-LOOPDONE)} \\ \mathbf{for} \ x \ \mathbf{in} \ v \sim: vs \ \{ t_2 \} \mid \mu &\longrightarrow \mathbf{let} \ x_1 = [x \mapsto v_1]t_2; \ \mathbf{for} \ x \ \mathbf{in} \ vs \ \{ t_2 \} \mid \mu && \text{(E-LOOPSTEP)} \\ &\text{where } x_1 \notin FV(t_2) \end{aligned}$$

In Section 5 and in our implementation, loops may in fact iterate over all data types that implement Rust's `Iterator` trait which for instance includes `HashSet`. This allows the developer to specify that the loop does not depend on a particular order and enables our second transformation that extracts amorphous data parallelism. To model this in \mathfrak{S}_s , we assume a stateless function that uses the iterator to collect the items into a list before looping over them. In fact, `collect` is a standard function of Rust's `Iterator`. We allow loops with an unknown iteration count via tail recursion. Tail recursion is a derived form:

¹⁰We allow *ConDRust* algorithms to be called from anywhere in a Rust program. Such a call may have arguments. The assumptions would be stated in Γ and require another context to access them during evaluation. We omit this detail at this point in favor of a concise presentation.

$$\boxed{\Delta, \Gamma \mid \Sigma \vdash t : T}$$

$$\frac{\Sigma(l) = T}{\Gamma \mid \Sigma \vdash l : \mathbf{Ref}\langle T \rangle} \text{T-LOC} \qquad \frac{}{\Delta, x : (1, T) \mid \Sigma \vdash x : T} \text{T-VAR}$$

$$\frac{\Delta, \Gamma_1 \mid \Sigma \vdash t_1 : T_1 \quad \Delta, \Gamma_2 \mid \Sigma \vdash t_2 : T_1 \rightarrow T_2}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma \vdash t_2(t_1) : T_2} \text{T-APPABS} \qquad \frac{x \notin \Gamma \quad \Gamma, x : (n, T_1) \mid \Sigma \vdash t : T_2}{\Delta, \Gamma \mid \Sigma \vdash | x : T_1 | \rightarrow T_2 \{ t \} : T_1 \rightarrow T_2} \text{T-ABS}$$

$$\frac{\Delta, \Gamma_1 \mid \Sigma_1 \vdash t_1 : T_1 \quad \Delta, \Gamma_2, x : (1, T_1) \mid \Sigma_2 \vdash t_2 : T_2}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma_1, \Sigma_2 \vdash \mathbf{let} x : T_1 = t_1; t_2 : T_2} \text{T-LET}$$

$$\frac{\Delta, \Gamma_1 \mid \Sigma_1 \vdash t_1 : T_1 \quad \Delta, \Gamma_2, x : (n, \mathbf{Ref}\langle \mathbf{mut} T_1 \rangle) \mid \Sigma_2 \vdash t_2 : T_2}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma_1, \Sigma_2 \vdash \mathbf{let mut} x : T_1 = t_1; t_2 : T_2} \text{T-LETMUT}$$

$$\frac{\Delta, \Gamma \mid \Sigma \vdash t_1 : T_1 \quad \Delta(\mathbf{f}_{SL}) = T_1 \rightarrow T_2}{\Delta, \Gamma \mid \Sigma \vdash \mathbf{f}_{SL}(t_1) : T_2} \text{T-FSL} \qquad \frac{\Delta, \Gamma_1 \mid \Sigma_1 \vdash t_1 : \mathbf{Ref}\langle \mathbf{mut} T_1 \rangle \quad \Delta, \Gamma_2 \mid \Sigma_2 \vdash t_2 : T_2 \quad \Delta(\mathbf{f}_{SF}) = \mathbf{mut} T_1 \rightarrow T_2 \rightarrow \circ T_3}{\Delta, \Gamma_1 \oplus \Gamma_2 \mid \Sigma_1, \Sigma_2 \vdash t_1.\mathbf{f}_{SF}(t_2) : T_3} \text{T-FSF}$$

$$\frac{\Delta, \Gamma_2 \mid \Sigma \vdash t_2 : \mathbf{Vec}\langle T_1 \rangle \quad \Delta, \Gamma_3, s : (1, \mathbf{Ref}\langle \mathbf{mut} T_s \rangle), x_1 : (n, T_1) \mid \Sigma \vdash t_3 : ()}{\Delta, \Gamma_2 \oplus \Gamma_3, s : (1, \mathbf{Ref}\langle \mathbf{mut} T_s \rangle) \mid \Sigma \vdash \mathbf{for} x_1 \mathbf{in} t_2 \{ t_3 \} : ()} \text{T-LOOP}$$

where $\forall s \in FV(t) \wedge s \neq x_1$

$$\frac{\Delta, \emptyset \mid \Sigma \vdash t_1 : (\mathbf{bool}, T_2, T_1) \quad \Delta, \emptyset \mid \Sigma \vdash t_2 : T_1 \rightarrow (\mathbf{bool}, T_2, T_1)}{\Delta, \emptyset \mid \Sigma \vdash \mathbf{trfix} t_1 t_2 : T_2} \text{T-FIX}$$

■ **Figure 24** *ConDRust*'s type system tracks and restricts variable usage.

$$\mathbf{let} f = | x : T_1 | \rightarrow T_2 \{
\begin{array}{l}
\mathbf{let} (x_1, x_2, x_3) = t_b; \\
\mathbf{if} x_1 \{ x_2 \} \\
\mathbf{else} \{ f(x_3) \}
\end{array}
\};$$

$$\stackrel{\text{def}}{=}$$

$$\mathbf{let} f = | x_0 : T_1 | \rightarrow T_2 \{
\begin{array}{l}
\mathbf{let} f' = | x : T_1 | \rightarrow (\mathbf{bool}, T_2, T_1) \{ t_b \}; \\
\mathbf{trfix} f'(x_0) f'
\end{array}
\};$$

Desugaring captures t_b , the computation of the recursion, as f' . Our tail-recursive combinator **trfix** takes two arguments. Argument 1 is an application f' to x_0 , the initial parameter of a recursive call. This application reduces to a triple (x_1, x_2, x_3) with the boolean discriminator x_1 , the final term x_2 and x_3 , the argument to the tail-recursive call. Argument 2 is f' for recursion. We encode the conditional that guards this tail recursive call into **trfix**'s semantics:

$$\mathbf{trfix} (\mathbf{true}, v_2, v_3) v_4 \mid \mu \longrightarrow v_2 \mid \mu \quad (\text{E-FIXDONE})$$

$$\mathbf{trfix} (\mathbf{false}, v_2, v_3) v_4 \mid \mu \longrightarrow \mathbf{trfix} v_4(v_3) v_4 \mid \mu \quad (\text{E-FIXRECUR})$$

In case the discriminator is **true**, we return the final result v_2 . Otherwise, we apply the recursive argument v_3 to the abstraction that is always the second argument of **trfix**. Again, we restrict the presentation to a single recursive argument and argue that the extension to multiple arguments is straightforward. In the context of this paper, we are particularly interested in the case where the arguments to the recursion are a state to be updated and a worklist that triggers these updates.

B.3 Type system

The *ConDRust* programming model carefully distinguishes between stateless and stateful computations. This enables the compiler to perform the translation of an algorithm into a dataflow representation and extract data parallelism while preserving the algorithm semantics.

The type system enforces the following programming discipline:

1. A variable may either be used as state or as input to a function call.
2. A variable that is input to a function call may only be used once.
3. A state variable may be used more than once except for a loop term where it may only be used once.

These somewhat restrictive rules are key enablers of our approach.

Data often needs to be shared across the concurrent parts of the program. A good example of this is the grid in the STM implementation that required a deep clone to make the concurrent execution scale. Often, introducing parallelism into a program represents a trade-off between speedup and memory efficiency. To make sophisticated decisions that strike a good balance for this trade-off, data structure knowledge is required by the compiler. We leave this to future work and enable the developer to explicitly make that decision by `cloneing` or sharing cloned `Arcs`. These techniques are already common practice for sharing data in Rust.

In *ConDRust*'s type system, presented in Figure 24, we use concepts from linear types to track and restrict variable usage [55]. The typing context Γ (defined in Figure 23) captures not only the types of variables but also their usage count (see rule T-VAR). The types T_{\oplus} are the types of the Rust programming language. With the rules T-LET and T-LETMUT, we distinguish between variables that are input to functions and state. In T-LET, *input variables* have a non-referential type and require a usage count of 1, i.e., they can only be used exactly once.¹¹ In T-LETMUT, *state variables* reference locations in the store μ and are marked with Rust's annotation for mutable types. Rules T-FSL and T-FSF specify the use of input and state variables in the type of the stateless and stateful function. Values in input position t_1 are of type T_1 while the state position t_s is required to be of type mutable reference $\text{Ref}\langle T_s \rangle$. We define the state encapsulation property of a stateful function on the output type as $\bigcirc T_3$. A type T has this property if it does not contain borrowed references.

The type information of the values behind the references is captured in the store typing Σ . As such, a typing judgement $\Delta, \Gamma \mid \Sigma \vdash t : T$ reads as follows:

► **Definition 4** (Well-typed). *Given an environment Δ and a (local variable) context Γ with type assumptions on store locations Σ ; a term t is well-typed if there exists a type T such that $\Delta, \Gamma \mid \Sigma \vdash t : T$.*

The rule T-LOOP restricts state variables to a single usage in the loop term t_3 . Only due to this restriction, the *ConDRust* compiler can derive pipeline parallelism. Rule T-FIX prevents tail recursive functions from accessing contextual variables at all by requiring $\Gamma = \emptyset$. Accessing captured variables in a recursive closure is also uncommon in Rust because the closure needs to explicitly communicate the lifetime of such variables. That is particularly challenging, for a closure that performs an unknown number of iterations.¹² In the spirit of

¹¹We treat unused variables as an undesirable property of a program that would benefit from a similar error message. Rust actually has similar warnings/errors for unused variables.

¹²Recursive closures need to be captured in structs to explicitly communicate lifetime information for the captured variables to the borrow checker. For more details see: <https://stevedonovan.github.io/rustifications/2018/08/18/rust-closures-are-hard.html>

$$\begin{aligned}
\Gamma \oplus \quad \quad \quad \emptyset &= \emptyset \\
\emptyset \oplus \quad \quad \quad \Gamma &= \Gamma \\
\Gamma_1, x : (n : T) \oplus \Gamma_2, x : (m : T) &= \Gamma_1 \oplus \Gamma_2, x : (n + m, T)
\end{aligned}$$

■ **Figure 25** Conjunction of typing contexts.

Terms	$t ::= x \mid v \mid n \mid c; t \mid \text{run}(t, t)$
Nodes	$n ::= \text{n}_{\text{SL}}(\text{f}_{\text{SL}}, t, t) \mid \text{n}_{\text{SF}}(\text{f}_{\text{SF}}, t, t, t, t) \mid \text{for}(t, t, t) \mid \text{reuse}(t, t, t) \mid \text{trfix}(t, t, t, t)$
Channels	$c ::= \text{let } x = \text{chan}(t) \mid \text{let } (x, x) = \text{chan}()$
Values	$v ::= l \mid v_{\oplus}$
Types	$T ::= \text{Ref}\langle T \rangle \mid T_{\oplus} \mid \text{mut } T_{\oplus}$
Typing context	$\Gamma ::= \emptyset \mid x : T$
Store	$\mu ::= \emptyset \mid \mu, l = v$
Store typing	$\Sigma ::= \emptyset \mid \Sigma, l : T$
Environment	$\Delta ::= \emptyset \mid \text{f}_{\text{SL}} : T \rightarrow T \mid \text{f}_{\text{SF}} : \text{mut } T \rightarrow T \rightarrow T$

■ **Figure 26** Syntactical constructs of \oplus_p .

linear types, we merge store typings via logical conjunction and define the conjunction for contexts in Figure 25. Based on this Rust subset, the *ConDRust* compiler can translate a sequential algorithm into a dataflow graph that makes all inherent parallelism explicit.

C \oplus_p – A subset of Rust for parallel composition

With the transformations in place, we now formally specify the backend of the *ConDRust* compiler to show that the generated code executes deterministically. We present the syntactic constructs for \oplus_p – a subset of Rust for parallel composition, that the *ConDRust* compiler targets in Figure 26. Terms in this subset basically consist of two parts:

Graph construction An arc is a channel (c) in Rust’s message-passing terminology and we define n , i.e., a term for each type of node in the dataflow graph.

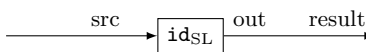
Graph execution We abstract over an explicit implementation of a scheduler for a dataflow graph with a single `run` construct.

We abstract over a concrete channel implementation. All we rely upon is the FIFO ordering property which we specify via the usual list constructors: $[\]$ empty list, $v \sim: v'$ (cons) where v is the head with the tail v' and the dual $v' \sim: v$ (snoc) where v is the last element in the list and v' the list of the preceding elements. Additionally, we assume the presence of tuples in the Rust values v_{\oplus} and types T_{\oplus} . Composition of nodes via arcs works solely via variable bindings. For example, the following term constructs a graph with a single (stateless) identity function (`idSL`) call node:

```

let src : Recv<i32> = chan(5);
let (result : Recv<i32>, out : Send<i32>) = chan();
run(result, (nSL(idSL, src, out) ~: [ ]))

```



For execution, we pass the receiving endpoint `result` and the list of nodes to `run` which executes the graph and reduces to the final result. We assume a type `Node` for nodes and align our specification for channels closely with `std::mpsc::channel` from Rust’s standard library where `Receiver<T>` and `Sender<T>` represent the receiving and the sending endpoint

$$\boxed{\Delta, \Gamma \mid \Sigma \vdash t : T}$$

$$\frac{\Sigma(l) = T}{\Gamma \mid \Sigma \vdash l : \mathbf{Ref}\langle T \rangle} \text{T-LOC} \quad \frac{}{\Delta, x : T \mid \Sigma \vdash x : T} \text{T-VAR}$$

$$\frac{\Delta, \Gamma_1 \mid \Sigma_1 \vdash t_1 : T_1 \quad \Delta, \Gamma_2, x : \mathbf{Recv}\langle T_1 \rangle \mid \Sigma_2 \vdash t_2 : T_2}{\Delta, \Gamma_1, \Gamma_2 \mid \Sigma_1, \Sigma_2 \vdash \mathbf{let } x = \mathbf{chan}(t_1); t_2 : T_2} \text{T-LETSRC}$$

$$\frac{\Delta, \Gamma, \begin{matrix} x_{11} : \mathbf{Recv}\langle T_{11} \rangle, \\ x_{12} : \mathbf{Send}\langle T_{12} \rangle \end{matrix} \mid \Sigma_1 \vdash t_2 : T_2}{\Delta, \Gamma \mid \Sigma_1, \Sigma_2 \vdash \mathbf{let } (x_{11}, x_{12}) = \mathbf{chan}(); t_2 : T_2} \text{T-LETCHAN}$$

$$\frac{\Delta(\mathbf{f}_{\text{SL}}) = T_1 \rightarrow T_2}{\Delta, \begin{matrix} x_1 : \mathbf{Recv}\langle T_1 \rangle, \\ x_2 : \mathbf{Send}\langle T_2 \rangle \end{matrix} \mid \Sigma \vdash \begin{matrix} x_1, \\ x_2 \end{matrix} : \mathbf{Node}} \text{T-NSL} \quad \frac{}{\Delta, \begin{matrix} x_1 : \mathbf{Recv}\langle \mathbf{Vec}\langle T \rangle \rangle, \\ x_2 : \mathbf{Send}\langle T \rangle, \\ x_3 : \mathbf{Send}\langle \mathbb{N} \rangle, \end{matrix} \mid \Sigma \vdash \begin{matrix} x_1, \\ x_2, \\ x_3 \end{matrix} : \mathbf{Node}} \text{T-NLP}$$

$$\frac{\Delta(\mathbf{f}_{\text{SF}}) = \mathbf{mut } T_4 \rightarrow T_1 \rightarrow T_2}{\Delta, \begin{matrix} x_1 : \mathbf{Recv}\langle T_1 \rangle, \\ x_2 : \mathbf{Send}\langle T_2 \rangle, \\ x_3 : \mathbf{Recv}\langle \mathbb{N}, T_3 \rangle, \\ x_4 : \mathbf{Send}\langle T_3 \rangle \end{matrix} \mid \Sigma \vdash \begin{matrix} x_1, \\ x_2, \\ x_3, \\ x_4 \end{matrix} : \mathbf{Node}} \text{T-NSF} \quad \frac{}{\Delta, \begin{matrix} x_1 : \mathbf{Recv}\langle T \rangle, \\ x_2 : \mathbf{Recv}\langle \mathbb{N} \rangle, \\ x_3 : \mathbf{Send}\langle \mathbb{N}, T \rangle \end{matrix} \mid \Sigma \vdash \begin{matrix} x_1, \\ x_2, \\ x_3 \end{matrix} : \mathbf{Node}} \text{T-NRU}$$

$$\frac{}{\Delta, \begin{matrix} x_1 : \mathbf{Recv}\langle T_1 \rangle, \\ x_2 : \mathbf{Send}\langle T_1 \rangle, \\ x_3 : \mathbf{Recv}\langle \mathbf{bool} \rangle, \\ x_4 : \mathbf{Recv}\langle T_2 \rangle, \\ x_5 : \mathbf{Recv}\langle T_1 \rangle, \\ x_6 : \mathbf{Send}\langle T_2 \rangle \end{matrix} \mid \Sigma \vdash \begin{matrix} x_1, \\ x_2, \\ x_3, \\ x_4, \\ x_5, \\ x_6 \end{matrix} : \mathbf{Node}} \text{T-NFIX} \quad \frac{\Delta, \Gamma \mid \Sigma \vdash t_2 : \mathbf{Vec}\langle \mathbf{Node} \rangle}{\Delta, \Gamma, x_1 : \mathbf{Recv}\langle T \rangle \mid \Sigma \vdash \mathbf{run}(x_1, t_2) : T} \text{T-RUN}$$

■ **Figure 27** The linear type system of \mathbb{S}_p for the construction and execution of the dataflow graph.

of a channel, respectively. In our encoding, the types $\mathbf{Recv}\langle T \rangle$ and $\mathbf{Send}\langle T \rangle$ are reference types (in $T_{\mathbb{S}}$) for locations l in the store μ , i.e., channels are values in the store. In fact, channels and their respective elements are the only values in the store. This simplification is possible because in *ConDRust* every state arrives along an arc, i.e., channel, where we preserve it across loop iterations. The context Γ (without usage tracking), the store μ , the store typing Σ and the environment Δ follow the specification in \mathbb{S}_s .

In the following, we first define the specifics of graph construction via the typing rules and afterwards present the operational semantics for graph execution to finally present our proof (sketch) for determinism.

C.1 Linear dataflow construction

Figure 27 defines the type system of \mathbb{S}_p . A dataflow graph consists of channels and nodes. In a dataflow graph each arc has exactly one sending node and one receiving node. To encode this invariant, we again resort to a linear type system approach and highlight linear aspects accordingly. The rules T-LOC and T-VAR type locations and variables. Channel construction is typed in rules T-LETSRC and T-LETCHAN. Source channels ($\mathbf{chan}(t)$) bind only a sending endpoint to pipe parameters from the surrounding Rust program into the dataflow graph. All other channels ($\mathbf{chan}()$) bind a receiving and a sending endpoint. In both cases, the usual conjunction of typing contexts (Γ_1, Γ_2) assures that each endpoint is

used exactly once. The rest of the rules (T-NSL, T-NSF, T-NLP, T-NRU, T-NFIX) concern the construction of nodes and the execution of the graph (T-RUN). We increase readability of the inference rules in the typing and evaluation relation in two ways. First, to make the flow of the different types of data more obvious, we highlight **receiving** and **sending** endpoints, **value reuse** and **state**. Second, to better align the type assumptions, we deliberately present the terms for nodes and **run** with variables instead of subterms, i.e., $n_{SL}(x_1, x_2)$ instead of $n_{SL}(t_1, t_2)$. This is not a restriction because we defined that channel endpoints have to be bound such that (node and **run**) terms requiring endpoint types can only be variables.

To keep the formal specification concise, only the T-NSF rules enables the reuse of data – in this case state. In the full formal specification and in our implementation, there are at least two versions for all nodes: one where the received input is of type T and another where it has an attached reuse count (\mathbb{N}, T) . Without loss of generality, we show this only for the state input of the stateful function call node n_{SF} .

C.2 Operational semantics

In the small-step operational semantics, we use the store μ to define the relation of nodes that can be executed. Evaluation is again from left to right as in \mathbb{S}_s following the E-CTXT rule from Figure 23. The construction of channels allocates dedicated locations in the store and the types **Send** and **Recv** are effectively references to a store location l .

$$\boxed{t \mid \mu \longrightarrow t' \mid \mu'}$$

$$\text{let } x_1 : \text{Recv}_{<_>} = \text{chan } v; t \mid \mu \longrightarrow [x_1 \mapsto l]t \mid \mu, l \mapsto (v \sim: []) \quad (\text{E-LETSRC})$$

where $l \notin \text{dom}(\mu)$

$$\text{let } (x_1 : \text{Recv}_{<_>}, x_2 : \text{Send}_{<_>}) = \text{chan}; t \mid \mu \longrightarrow [x_1 \mapsto l, x_2 \mapsto l]t \mid \mu, l \mapsto [] \quad (\text{E-LETCHAN})$$

where $l \notin \text{dom}(\mu)$

For a source channel, the corresponding value is stored directly into the list and only the receiving end is emitted. Hence, the channel emits exactly one data value and remains empty for the rest of the computation. The receiving end x_1 and the sending end x_2 for all other channels, point to the **same location** l in the store. A channel is initially an empty list. For both types of channels, the evaluation makes a step using beta reduction.

The interesting part is the execution of the dataflow graph. Computation is complete when there is a value in the channel of the final endpoint.

$$\frac{}{\text{run}(v, l) \mid \mu, l \mapsto (v \sim: []) \longrightarrow v \mid \mu} \text{E-DONE} \quad \frac{\exists v_n \in v_g. v_n \mid \mu \longrightarrow v_n \mid \mu'}{\text{run}(v_g, l) \mid \mu, l \mapsto [] \longrightarrow \text{run}(v_g, l) \mid \mu'} \text{E-RUN}$$

Otherwise, there must a node v_n in the list of nodes v_g that can take a step that updates the store μ . This property holds because our linear typed construction does not allow for dangling channels and the following rules for the nodes in the graph always send data in every step. A stateless call node can make a step if its incoming channel referenced by label l_1 has data available:

$$\frac{\mathbf{f}_{SL}(v_1) \mid \emptyset \Downarrow_{\mathbb{S}} v'_1 \mid \emptyset}{n_{SL}(\mathbf{f}_{SL}, l_1, l_2 \mid \mu, \begin{matrix} l_1 \mapsto (v_1 \sim: v_{12}), \\ l_2 \mapsto v_2 \end{matrix}) \longrightarrow n_{SL}(\mathbf{f}_{SL}, l_1, l_2 \mid \mu, \begin{matrix} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 \sim: v'_1) \end{matrix})} \text{E-NSL}$$

The node retrieves the head v_i of the channel's list, performs the call and appends the resulting data value v'_1 to the list of the outgoing channel (l_2). The execution of the stateless function itself is the same as defined in the operational semantics for stateless calls (E-FSL) in \mathbb{S}_s defined in Section 4. Loop nodes follow the same execution pattern:

$$\frac{v_1.\text{size}() \mid \emptyset \Downarrow_{\mathbb{S}} n \mid \emptyset}{\text{for}(l_1, l_2, l_3) \mid \mu, \begin{array}{l} l_1 \mapsto (v_1 \sim: v_{12}), \\ l_2 \mapsto v_2, \\ l_3 \mapsto v_3 \end{array} \longrightarrow \text{for}(l_1, l_2, l_3) \mid \mu, \begin{array}{l} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 \sim: v_1), \\ l_3 \mapsto (v_3 \sim: n) \end{array}} \text{E-NLOOP}$$

A loop node streams the incoming list v_1 by concatenating ($\sim:$) it with the outgoing channel's list v_2 . We define list concatenation as usual:

$$xs \sim: (y_1 \sim: (y_2 \sim: \dots (y_n \sim:))) = (\dots ((xs \sim: y_1) \sim: y_2) \dots \sim: y_n)$$

The loop node additionally emits the number of streamed elements n to a reuse node that pairs it with the gated value:

$$\text{reuse}(l_1, l_2, l_3) \mid \mu, \begin{array}{l} l_1 \mapsto (v_1 \sim: v_{12}), \\ l_2 \mapsto (v_2 \sim: v_{22}), \\ l_3 \mapsto v_3 \end{array} \longrightarrow \text{reuse}(l_1, l_2, l_3) \mid \mu, \begin{array}{l} l_1 \mapsto v_{12}, \\ l_2 \mapsto v_{22}, \\ l_3 \mapsto (v_3 \sim: (n, v_1)) \end{array} \quad (\text{E-NREUSE})$$

In our presentation, only state is reused. To do so, the stateful function node receives the state value v_2 with the attached reuse count n on its state channel l_3 .

$$\frac{l_s.\text{f}_{SF}(v_1) \mid l_s \mapsto v_3 \Downarrow_{\mathbb{S}} v_{22} \mid l_s \mapsto v'_3}{\text{n}_{SF}(\text{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{l} l_1 \mapsto (v_1 \sim: v_{12}), \\ l_2 \mapsto v_2, \\ l_3 \mapsto ((n, v_3) \sim: v_{31}), \\ l_4 \mapsto v_4 \end{array} \longrightarrow \text{n}_{SF}(\text{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{l} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 \sim: v_{22}), \\ l_3 \mapsto ((n-1, v'_3) \sim: v_{31}), \\ l_4 \mapsto v_4 \end{array}} \text{E-NSFREUSE}$$

$$\frac{l_s.\text{f}_{SF}(v_1) \mid l_s \mapsto v_3 \Downarrow_{\mathbb{S}} v_{22} \mid l_s \mapsto v'_3}{\text{n}_{SF}(\text{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{l} l_1 \mapsto (v_1 \sim: v_{12}), \\ l_2 \mapsto v_2, \\ l_3 \mapsto ((1, v_3) \sim: v_{31}), \\ l_4 \mapsto v_4 \end{array} \longrightarrow \text{n}_{SF}(\text{f}_{SF}, \begin{array}{l} l_1, \\ l_2, \\ l_3, \\ l_4 \end{array}) \mid \mu, \begin{array}{l} l_1 \mapsto v_{12}, \\ l_2 \mapsto (v_2 \sim: v_{22}), \\ l_3 \mapsto v_{31}, \\ l_4 \mapsto v_4 \sim: v'_3 \end{array}} \text{E-NSFEMIT}$$

Rule E-NSFREUSE preserves the computed state v'_3 with a decremented reuse count in the *incoming* state channel l_3 . When the reuse count is 1 then Rule E-NSFEMIT emits v'_3 to the outgoing state channel. Both rules rely on $\Downarrow_{\mathbb{S}}$ which requires the state value v_3 to be behind a reference l_s . But l_s exists solely to satisfy this requirements. It is not contained anymore in our store μ . That is the result of the translation from an imperative into a functional program in the *ConDRust* compiler. The `trfix` node (which we omit at this point for brevity) follows the same principle to wait for a single recursive call to complete: It enqueues a \perp data value into its incoming arc x_1 that is dequeued only when a recursion is finished and the resulting value was sent along the respective channel x_6 (see T-NFIX).

C.3 Determinism

With the operational semantics and typing relation defined, we can prove that evaluation in \mathbb{S}_p and as such execution in *ConDRust* is deterministic. The evaluation relation keeps the selection of the next node abstract. It just states the particular data availability requirements

for the nodes to be evaluated. Stateful call nodes have more than one outgoing channel effectively creating subgraphs, i.e., task-level parallelism, and potentially adding more than a single successor into the evaluation relation. Additionally, a for-node emits a whole stream of data values allowing the downstream nodes to be executed repeatedly, i.e., in a pipeline parallel fashion. As such, there may be more than one node ready to be evaluated. The evaluation relation does not specify a concrete evaluation order and applies to any scheduler that follows the defined evaluation rules. As such, we show that evaluation in $t \mid \mu \longrightarrow t' \mid \mu'$ is deterministic.

► **Lemma 5 (Single-step Determinism).** *If $t : T$ is a well-typed term in \mathbb{S}_p then*

$$\begin{aligned} t \mid \mu, l \mapsto vs &\longrightarrow t' \mid \mu', l \mapsto (v \sim: vs) \quad \wedge \\ t \mid \mu, l \mapsto vs &\longrightarrow t'' \mid \mu'', l \mapsto (v \sim: vs) \Rightarrow t' = t'' \wedge \mu' = \mu''. \end{aligned}$$

Proof Sketch. The proof is by induction on a derivation of t and the store μ . Assume a term t' whose activation into the evaluation relation requires data value v available at store location l . For a term t whose evaluation places v into store location l , we distinguish the following two cases:

1. t is the construction of a source channel or
2. t is the evaluation of an upstream node.

The first case is immediate. In the second case, we also know that by the induction hypothesis the activations of this (upstream) node is deterministic. Now assume that t evaluates to t'' by storing a value at location l . By the linear construction of the channels in the dataflow graph, store location l only has a single receiver that is owned by exactly one node. Hence, $t' = t''$ and consequently $\mu' = \mu''$. ◀

Dataflow graphs in \mathbb{S}_p are essentially Kahn Process Networks (KPN) [31]. KPNs execute deterministically because incoming arcs have blocking semantics¹³ and the executed code of the node is scott-continuous. Our evaluation relation essentially adheres to both of these properties.

However, lemma 5 is insufficient to prove determinism for the whole computation, i.e., for the multi-step evaluation $t \mid \mu \longrightarrow^* v \mid \mu'$:

► **Theorem 6 (Determinism).** *If $t : T$ is a well-typed term with $t_s \xrightarrow{\text{ConDRust}} t$ then*
 $t \mid \mu \xrightarrow{c}^* v \wedge t \mid \mu \xrightarrow{c}^* v' \Rightarrow v = v'$.

The proof of this theorem needs a proof of termination which in turn requires two things: well-founded recursion and cycle freedom. Cycle freedom is based on a formal specification of the transformations in *ConDRust* to prove that the dataflow graph does not have cycles other than the ones guarded by `trfix` nodes. This formalization is outside the scope of this paper and left for future work.

¹³Blocking semantics prevent the construction of a non-deterministic merge node, the explicit notion of non-determinism in dataflow [2].

Dependent Merges and First-Class Environments

Jinhao Tan ✉

The University of Hong Kong, China

Bruno C. d. S. Oliveira ✉

The University of Hong Kong, China

Abstract

In most programming languages a (runtime) *environment* stores all the definitions that are available to programmers. Typically, environments are a meta-level notion, used only conceptually or internally in the implementation of programming languages. Only a few programming languages allow environments to be *first-class values*, which can be manipulated directly in programs. Although there is some research on calculi with first-class environments for statically typed programming languages, these calculi typically have significant restrictions.

In this paper we propose a statically typed calculus, called E_i , with first-class environments. The main novelty of the E_i calculus is its support for first-class environments, together with an expressive set of operators that manipulate them. Such operators include: *reification* of the current environment, environment *concatenation*, environment *restriction*, and *reflection* mechanisms for running computations under a given environment. In E_i any type can act as a context (i.e. an environment type) and contexts are simply types. Furthermore, because E_i supports subtyping, there is a natural notion of context subtyping. There are two important ideas in E_i that generalize and are inspired by existing notions in the literature. The E_i calculus borrows *disjoint intersection types* and a *merge operator*, used in E_i to model contexts and environments, from the λ_i calculus. However, unlike the merges in λ_i , the merges in E_i can depend on previous components of a merge. From implicit calculi, the E_i calculus borrows the notion of a *query*, which allows *type-based* lookups on environments. In particular, queries are key to the ability of E_i to reify the current environment, or some parts of it. We prove the determinism and type soundness of E_i , and show that E_i can encode all well-typed λ_i programs.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases First-class Environments, Disjointness, Intersection Types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.34

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*:
<https://doi.org/10.4230/DARTS.9.2.2>

Funding Hong Kong Research Grant Council projects number 17209520 and 17209821 sponsored this work.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

In most programming languages, (runtime) *environments* are used to store all the available definitions at a given point in a program. Typically, an environment is a dictionary that maps variable names to values. However, environments are normally a meta-level concept, which does not have any syntactic representation in source programs. Environments may be used internally in the implementation of programming languages. For example, in implementing functional languages, closures are often used to keep the lexical environment of a function around. However, it is impossible for programmers to write directly a closure or manipulate environments explicitly.



© Jinhao Tan and Bruno C. d. S. Oliveira;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 34; pp. 34:1–34:32



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



First-class environments [18, 23, 24, 29, 36], are environments that can be created, composed, and manipulated at runtime. In programming languages with first-class environments, programs have an explicit syntactic representation for environments that enables them to be first-class values. As argued by Gelernter et al. [18], with first-class environments, the distinction between declarations and expressions can be eliminated. Furthermore many programming language constructs – including *closures*, *modules*, *records* and *object-oriented constructs* – can be modelled with first-class environments. However, only a few programming languages allow environments to be first-class values. These languages are mainly dynamically typed languages such as dialects of Lisp [18] and the R language [17]. Typically, operations on environments include: reification (transforming environments into data objects), reflection (treating data objects as environments), environment restriction (returning part of an environment), and environment composition/concatenation. While dynamically typed languages with first-class environments give users high flexibility in manipulating environments, several runtime type errors are unavoidable due to the absence of static typing.

Compared with work on dynamically typed languages, there is much less research on statically typed languages with first-class environments [44, 45, 47]. In these works, environments as first-class values have a special kind of type which is called an *environment type*. Although static typing prevents some of the runtime errors, subtyping is not included in existing type systems with environment types. At the term level, there are two constructs for environments: one is an evaluation operation $e[[a]]$ that evaluates the expression a under an environment e , and the other is an operator returning the current environment. While these two constructs model importing and exporting of environments respectively, there are no facilities for concatenation or restriction of environments in these calculi.

In this paper we propose a statically typed calculus, called E_i , with first-class environments. The main novelty of the E_i calculus is its support for first-class environments with an expressive set of operators that manipulate first-class environments. In E_i , both reflection mechanisms for running computations under a given environment and reification of the current environment are supported. Moreover, compared with previous work on typed calculi with first-class environments, environment *concatenation* and environment *restriction* are allowed. In E_i , types and contexts (i.e. environment types) are completely unified. That is, any type can act as a context and contexts are simply types. Unlike previous calculi, E_i also supports subtyping and has a natural notion of subtyping of environments. In E_i , users can benefit from static typing for handling type errors at compile-time, while still having various flexible mechanisms to manipulate environments.

In order to model environments, the E_i calculus borrows *disjoint intersection types* and the *merge operator* from the λ_i calculus [32]. The novelty in E_i is to additionally use intersection types to model environment types (or contexts), and disjointness to model disjointness/uniqueness of variables in an environment. Correspondingly, in E_i , the merge operator enables constructing and concatenating environments. Moreover, unlike λ_i , the merges in E_i can depend on previous components of a merge. In other words, merges in E_i are *dependent* (note that the dependency in a merge is term-level dependency, and it should not be confused with dependent types). Unifying contexts and types enables type information flowing from the left branch to the right branch in a merge, such that the type of the left branch becomes part of the context of the right branch. Consequently, with reification, the right branch of a merge can construct an expression based on the left branch of a merge. For example, the following program (with syntactic sugar)

$$\{x = 1\}, \{y = x\}$$

is well-typed in E_i . Here y in the right branch can access x and build a value under the environment $\{x = 1\}$. The merge will be evaluated to $\{x = 1\} \circ \{y = 1\}$. Dependent merges are useful for modelling dependent declarations, which are not expressible in λ_i since a field in a single record cannot access the field in a previous record in a merge.

Instead of looking up values by names as in traditional lambda calculi, the E_i calculus borrows the notion of a *query*, which enables *type-based* lookups on environments, from *implicit calculi* [12,31,46]. In implicit calculi, queries are used to query *implicit* environments by type. However, in E_i , queries are applied directly to runtime environments instead, and they are key to the ability of E_i to reify the current environment, or some parts of it. Effectively, a query can synthesize the current context (in typing) and the current environment (during reduction). With type annotations, queries can choose part of the environment based on those annotations, modelling environment restriction.

In our work, we prove the determinism and type soundness of E_i , and show that E_i can encode all λ_i programs. The E_i calculus and all the proofs presented in this paper have been formalized in the Coq theorem prover [9]. In summary, the contributions of this paper are:

- **Dependent merges as first-class environments:** We propose the novel notion of dependent merges, which allow dependencies appearing in merges. With dependent merges, dependent declarations and first-class environments can be modelled easily in a natural way.
- **The E_i calculus:** We present a statically typed calculus called E_i with support for creation, reification, reflection, concatenation and restriction of first-class environments. In addition, we study an extension with fixpoints (shown in the appendix). Both calculi are deterministic and type sound.
- **Encoding of the λ_i calculus:** We show that E_i can encode the type system of the λ_i [32] via a type-directed translation. In other words, standard variables, lambda abstractions, and non-dependent merges can be fully encoded in E_i .
- **Coq formalization:** All the results presented in this paper have been formalized in the Coq theorem prover and they are available in the artifact associated to this paper: <https://github.com/tjhao/ecoop2023>

2 Overview

This section gives an overview of our work. We start with some background on the merge operator, first-class environments and program fragments. Then we discuss challenges of modelling first-class environments as merges and finally we discuss the key ideas in our work.

2.1 Background

The merge operator and disjoint intersection types. The original non-dependent merge operator (denoted here by \circ) was firstly introduced by Reynolds [38] and later refined by Dunfield [15]. Merges add expressiveness to terms, enabling constructing values that inhabit intersection types. Essentially, with the merge operator, values are allowed to have multiple types. For example, the following program is valid:

```
let x : Bool & Int = true , 1 in (not x, succ x)
```

In the program above, the variable x has types `Bool` and `Int`, encoded by the intersection type `Bool & Int`. At the term level, x is created with the merge operator and can be regarded as either a boolean or an integer when used. For instance, in the program above there are two uses of x , one as a boolean (as the argument to `not`) and one as an integer (as the argument to `succ`). A language with the merge operator is able to extract the value of the right type

34:4 Dependent Merges and First-Class Environments

■ **Table 1** Summary of common operators on environments. E denotes an environment, I denotes a set of identifiers, and $T1$ and $T2$ denote terms in the language.

Operator	Description
<code>export</code>	Exports/reifies the full <i>current</i> environment.
$E \setminus I$	Returns a new <i>restricted</i> environment that only contains the identifiers in I from the environment E .
<code>import(T1, T2)</code>	Evaluates $T1$ to be an environment $E1$, and uses $E1$ to evaluate $T2$.
<code>import(I, T1, T2)</code>	Evaluates $T1$ to be an environment $E1$, checks that a set of identifiers I are defined in $E1$, then uses $E1$ to evaluate $T2$.
$E1, E2$	Composes/concatenates two environments.

from merges. In many classical systems with intersection types, but without the presence of the merge operator, the type `Bool & Int` cannot be inhabited and the program above is not expressible [34].

An important issue that the merge operator introduces is ambiguity. What happens if merges contain multiple values of the same type? For example, we could have $(1, 2) : \text{Int}$, but if this is allowed, then it could result in either 1 or 2. To address the ambiguity problem, Oliveira et al. [32] presented the λ_i calculus, which imposed a restriction where only merges of values that have *disjoint types* are accepted (we use $A * B$ to represent that A is disjoint with B). In this way, ambiguous programs such as $1, 2$ are rejected since `Int` is not disjoint with itself. However, `Bool` and `Int` are disjoint, and thus `true, 1` is a well-typed expression.

As Dunfield [15] argued, with the merge operator, many language features such as *dynamic typing*, *multi-field records*, and *operator overloading* can be easily encoded. After that, several non-trivial programming language features, including *dynamic mixins* [2], *first-class traits* [5], *nested composition* [6, 22] have been enabled with the help of the merge operator and disjoint intersection types. These features provide the foundations for *compositional programming* [51], which is a programming paradigm that enables a simple and natural solution to the Expression Problem [49] and other modularity problems. Compositional programming is realized in the CP language [51], which has been used to demonstrate the expressive power of the paradigm.

First-class environments. Normally, environments are not a syntactic entity of a programming language. Instead, environments exist implicitly at the meta-level for defining formal semantics and implementing languages. However, some dynamically typed languages, including dialects of Lisp [18] or the R language [17], include support for first-class environments. There is a line of research work on first-class environments for dynamically typed languages [18, 23, 24, 29, 36]. First-class environments provide a lot of expressive power, and they are used to model many other language constructs. With first-class environments, it is possible to model *closures*, *modules*, *records* or *object-oriented constructs* [18]. Moreover, it is also possible to model declarations directly, eliminating the need to distinguish between declarations and expressions.

To allow environments manipulated by not only compilers or interpreters but also programmers, a form of *reification* and *reflection* of environments is needed. Reification transforms environments into data objects and reflection enables data objects to be treated as environments [23, 24]. While formalizations differ, generally speaking, environments are formalized as a mapping from variables to data objects, which can be manipulated at runtime. We summarize typical supported operators to manipulate environments [36] in Table 1 (with notations slightly changed).

Work on first-class environments for typed languages [44, 45, 47] comes with significant restrictions compared to what is supported in dynamically typed languages. In these calculi, types and environment types are defined such that environment types are a special kind of type. The definition of types is $A, B ::= A \rightarrow B \mid \dots \mid E$, and each environment type E has the form of $\{x_1 : A_1, \dots, x_m : A_m\}$ where A_i ($1 \leq i \leq m$) is a type and each variable x_i must be distinct (or disjoint) with each other. Environment types encode exactly the normal typing context, which is a set that consists of typing assumptions $x_i : A_i$. Correspondingly, an environment has the form of $\{a_1/x_1, \dots, a_m/x_m\}$ that binds variables x_i with terms a_i [44, 47]. There are two constructs related to environments:

- The first construct returns the current environment which acts similarly to `export`.
- The second construct is an evaluation operation $e[[a]]$ that evaluates the expression a under an environment e . Note that, this operation is similar to `import(T1, T2)` in Table 1 (where `T1` corresponds to e and `T2` corresponds to a).

With these two constructs, one can create an environment at run-time and use it for evaluation. However, types are not totally unified with environment types in this setting, which results in special treatment of environments. For example, the expression e in $e[[a]]$ can only be an environment. To avoid runtime errors, the typing rule for $e[[a]]$ restricts the type that e has to be an environment type. Existing type systems with environment types do not consider subtyping. At the term level, though environments can be computed by evaluation under other environments and function applications, concatenation or restriction of environments are not supported. Therefore, an environment with a larger/smaller width cannot be constructed on the fly either. In short, there is no subtyping and the operations that are supported in dynamically typed languages in Table 1 are not fully supported in typed calculi with first-class environments.

Program fragments and separate compilation. To motivate our work we will show how first-class environments can be helpful to model a simple form of modules. Our form of modules is inspired by Cardelli’s [7] *program fragments*. Here we first introduce the notion of program fragments, and in Section 2.3 we will see how we can model program fragments in E_i .

A *program fragment*, or *module*, is a syntactically well-formed expression where free variables may occur [7]. *Separate compilation* decomposes a program into program fragments that can be typechecked and compiled separately. A program fragment may contain free variables. However, if the required interface that contains adequate type information is specified, then the types of the free variables can be found (without any concrete implementation). Thus, the typechecking of a program fragment can still be carried out *separately*.

In a conventional calculus, such as the simply typed lambda calculus (STLC), we express abstractions over a variable annotated with a type. However, there are no facilities for abstracting over an interface that may consist of multiple (nested) type assumptions. In other words, the STLC is not powerful enough to model separate compilation.

Cardelli [7] proposed a calculus of program fragments for the STLC, and specified high-level abstractions for modules and interfaces. In Cardelli’s framework, interfaces are interpreted as typing contexts that are *external* to the language. A module that may require an interface/context is represented as a binding judgment $E \vdash d \cdot S$, where E is a context, d a list of definitions, S a list of type declarations. Take the following modules from Cardelli as an example:

34:6 Dependent Merges and First-Class Environments

```
module
import nothing
export x: Int
begin
  x : Int = 3
end.

module
import x: Int
export f: Int → Int, z: Int
begin
  f : Int → Int = λ(y: Int). y+x
  z : Int = f(x)
end.
```

These two modules can be modelled as two binding judgments:

$$\emptyset \vdash (x : \text{Int} = 3) \therefore (x : \text{Int})$$
$$x : \text{Int} \vdash (f : \text{Int} \rightarrow \text{Int} = \lambda(y : \text{Int}). y + x, z : \text{Int} = f(x)) \therefore (f : \text{Int} \rightarrow \text{Int}, z : \text{Int})$$

A module is encoded as a list of definitions d , with an import list modelled as a context E and an export list as type declarations S . In the second module, z relies on f . To model such dependency, the binding judgment $E \vdash d \therefore S$ is designed to be *dependent*: each component depends on its previous components in d , in the sense that every free variable in this component can refer to its corresponding type. To check whether $z : \text{Int} = f(x)$ is matched by $z : \text{Int}$, the type declaration $f : \text{Int} \rightarrow \text{Int}$ is appended to the original context $x : \text{Int}$ to be a type assumption. In this way, the second binding judgment can be checked separately since each variable can access sufficient type information.

Though each binding judgment can be separately compiled to a self-contained entity called a linkset, user-defined abstractions cannot be expressed in Cardelli's work, since a binding judgment itself is a meta-level notion that cannot be created by programmers. In our work, we also regard interfaces as typing contexts. However, we unify typing contexts and types, and there are *first-class* constructs that abstract over a type/interface. We will discuss our ideas in detail in Section 2.3.

2.2 Limitations of Non-Dependent Merges

As Section 2.1 argued, both the (non-dependent) merge operator and first-class environments are useful to model a variety of other language constructs. Some of these language constructs can even be modelled by both merges or first-class environments. Given the overlap between merges and first-class environments it is reasonable to try to unify them, to obtain a more powerful model of statically typed languages with first-class environments. Our goal is to use merges to model first-class environments. However, non-dependent merges in existing calculi such as λ_i are inadequate for this purpose. This section discusses the limitations of non-dependent merges that are addressed by us.

No support for reification and reflection of environments. Intersection types and the merge operator are powerful tools that enable many language features, one of which is multi-field records [40]. In fact, multi-field record types can be turned into an intersection of single-field record types:

$$\{l_1 : A_1, \dots, l_n : A_n\} \equiv \{l_1 : A_1\} \& \dots \& \{l_n : A_n\}$$

Recall the syntax of conventional typing contexts: $\Gamma ::= \cdot \mid \Gamma, x : A$. A typing context is a list of pairs that bind variables with types. If we view variables as labels, typing contexts can be encoded as multi-field records, which are further desugared to intersections of single-field record types. Similarly, at the term level, a multi-field record is expressed as a merge of single-field ones:

$$\{l_1 = e_1, \dots, l_n = e_n\} \equiv \{l_1 = e_1\}, \dots, \{l_n = e_n\}$$

For example, $\{x = 2, y = 4\}$ is encoded as $\{x = 2\}, \{y = 4\}$. In calculi with a merge operator, merges are always *first-class* expressions and thus they can be passed to functions.

However, in previous calculi with the merge operator [15, 32, 38], merges are not used to model environments. Therefore, there are no reification and reflection facilities for environments in those calculi. Furthermore, intersection types are not used to model contexts, and there is no construct that enables running some computation under a local environment. In short, previous calculi with the merge operator support concatenation, but they do not support other operations in Table 1.

No dependent merges. An important limitation of merges in previous work with respect to environments is that they cannot be dependent. Many programming languages, as well as Cardelli’s program fragments, support declarations such as:

```
let x = 2
let y = 4
```

which allows several declarations to be associated with expressions. For the declarations above, we can easily model them as a (non-dependent) merge of two single field records:

```
{x = 2}, {y = 4}
```

where variables x and y are encoded as field names (or labels), and the values assigned to variables are modelled as record fields.

The previous declarations are *non-dependent*, in the sense that the expression assigned to y does not refer to x . However, in practice many declarations are *dependent*, where the current declaration relies on previous ones. For instance, fairly often we may have a program:

```
let x = 2
let y = x + x
let main = x + y
```

where y depends on x and main depends on both y and x . The traditional non-dependent merge operator cannot capture such cases. To be concrete, consider the typing rule for merges from λ_i [32]:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \vdash e_2 \Rightarrow B \quad A * B}{\Gamma \vdash e_1, e_2 \Rightarrow A \& B} \text{TYP-MERGE}$$

Here $A * B$ expresses that A and B are disjoint types. For typing a merge e_1, e_2 , the typing context for the right branch e_2 is Γ and does not contain any type information about the left branch e_1 . When typing e_2 , the type of e_1 is never used during the typing procedure. As a result, e_2 cannot be built by referring to e_1 . Moreover, to cooperate with the static semantics, the two branches of e_1, e_2 are evaluated separately without dependency involved in the dynamic semantics of λ_i . That is, the environment for evaluating e_2 does not contain the evaluation result of e_1 .

The incapacity of encoding dependent declarations as first-class expressions exposes that λ_i is not able to fully model *module*-related language features. Since dependent definitions/declarations often occur in a module, as shown in our discussion on program fragments in Section 2.1.

2.3 Key Ideas

In this work, we utilize the merge operator together with new constructs to enable dependent merges and first-class environments. We concretize these ideas in a new calculus called E_i . The key ideas of our work are discussed next.

Typing contexts as types. In E_i the typing context in the typing judgment is a *type* instead of an association list. Our grammar for *both* types and contexts is:

$$A, B, \Gamma ::= \text{Int} \mid \text{Top} \mid A \rightarrow B \mid A \& B \mid \{l : A\}$$

A typing assumption $x : A$ in conventional calculi is modelled as a record type, and the intersection of two types plays a similar role to the concatenation of two association lists. However, the contexts in our work are not restricted to intersections of record types. In fact, any type (e.g., Int) defined in the syntax of E_i can be a typing context. If a context consists only of Top (the top type), then there is no type information in this context, which corresponds to an empty association list. As we will see, viewing typing contexts as types opens up the possibility of creating interesting language features.

Unifying environments and expressions. Just as typing contexts are types in E_i , environments in the reduction semantics are just *values* instead of association lists which bind variable names to values. Hence, environments are *first-class* in our setting. The top value \top is used to model the empty environment. A merge of two values can be viewed as concatenation of two environments. For example, the merge $\{x = 1\} \circ \{y = 2\}$ is a valid environment that binds 1 and 2 to x and y respectively. In E_i , we denote the merge operator by a single comma (\circ) to follow the notation conventionally used in programming languages to denote the concatenation of two environments. With record projection, the value bound to a label can be accessed. Note that unifying environments and values and viewing variables as labels means that extra syntax (or data structures) for environments is not needed. This is different from previous work on typed calculi with first-class environments where an explicit notion of environments is introduced [44, 45, 47].

In E_i , we have two constructs to support reification (or exporting) and reflection (or importing) of environments. For reification, we employ the *query* construct $?$. The query construct is inspired by the implicit calculus [12], where queries are used to query *implicit* environments by type. In E_i we apply queries directly to runtime environments instead, whereas in the implicit calculus, access to the regular environments is done conventionally using named variables. The typing rule for $?$ is simply:

$$\Gamma \vdash ? \Rightarrow \Gamma$$

i.e. the query $?$ synthesizes the current context. For example, $\{x : \text{Int}\} \vdash ?.x \Rightarrow \text{Int}$ is valid. Here $?$ obtains the current environment and accesses the field x .

Regarding the reflection of environments, there is a construct $e_1 \triangleright e_2$ that is called *box* in E_i . In a box, e_2 is assigned an expression e_1 , which is evaluated to be a value that acts as an environment for evaluating e_2 . Take $\{x = 1 + 1\} \triangleright ?.x + 1$ as an example. The expression $\{x = 1 + 1\}$ is given as the environment to $?.x + 1$. Then $\{x = 1 + 1\}$ is evaluated to $\{x = 2\}$, under which $?.x + 1$ is evaluated to 3. The box construct can be seen as the inverse operator of the query, since $? \triangleright e$ is equivalent to e in the sense that $?$ exports the full environment by default. Allowing e_1 in the box $e_1 \triangleright e_2$ to be any well-typed expression instead of a value adds expressiveness to reflection. For example, environment injection can be encoded as $(? \circ v) \triangleright e$ where v is added to the original environment for e *locally*.

In E_i , type annotations play a role in information hiding. For example, for a merge with an annotation $(\{x = 1\}, \{y = 2\}) : \{x : \text{Int}\}$, only $\{x = 1\}$ is visible. Type annotations provide a mechanism to enable restriction, since they are able to prevent visibility of certain values. Since environments are values in our setting, type annotations can seal the environment, such that only components named in the type are accessible. Therefore, reification and reflection are *type-directed* in E_i . With type annotations, users can choose part of the environment that they desire. In summary, E_i can essentially model all the operations on environments in Table 1 with the following expressions:

- $?$ reifies the entire environment;
- $? : A$ obtains part of the environment that has type A ;
- $e_1 \triangleright e_2$ evaluates e_1 to an environment and uses that to evaluate e_2 under that environment;
- $(e_1 : A) \triangleright e_2$ evaluates e_1 , but restricts the resulting environment to A and uses that to evaluate e_2 ;
- e_1 , e_2 concatenates two environments e_1 and e_2 .

Dependent merges. To model dependent declarations, the merges in our work are dependent. The right branch can refer to the type of the left branch. The typing rule for dependent merges is:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \& A \vdash e_2 \Rightarrow B \quad A * \Gamma \quad A * B}{\Gamma \vdash e_1 , e_2 \Rightarrow A \& B} \text{ TYP-DMERGE}$$

Modelling *typing contexts as types* enables type information flowing from the left branch to the right branch in a merge. Specifically, for e_1 , e_2 , the type of e_1 is added into the current context such that e_2 synthesizes a type under the intersection type $\Gamma \& A$.

Suppose that the current context Γ is a subtype of some type B , with type annotation, $? : B$ exports B from Γ . With the query construct, a dependent declaration can be encoded as a dependent merge:

$$\{x = 2\} , \{y = (? : \{x : \text{Int}\}) . x + (? : \{x : \text{Int}\}) . x\}$$

which has type $\{x : \text{Int}\} \& \{y : \text{Int}\}$. The annotated query $? : \{x : \text{Int}\}$ exports $\{x : \text{Int}\}$ from the context, and projection $(? : \{x : \text{Int}\}) . x$ infers the type Int . Note that E_i is meant as a minimal core calculus and it is not built with convenience in mind. So the expression above is more cumbersome than what a programmer would expect to write in a source language. With some basic support for type inference and syntactic sugar in a source language, we could write instead:

$$\{x = 2\} , \{y = ? . x + ? . x\}$$

or even:

$$\{x = 2\} , \{y = x + x\}$$

In Section 5 we show how some of this syntactic sugar and inference can be achieved. For readability purposes, in the following examples, we will take the liberty to use a more lightweight syntax for the examples written in E_i as well.

In general, dependent declarations can be modelled as a merge of expressions e_1 , \dots , e_n , where the type information accumulates from e_1 to e_n . Modelling declarations as merges means that while we can benefit from the expressiveness of the merge operator, we do not need to introduce an additional syntax for declarations. Besides the condition $A * B$ that avoids conflicts between two branches in a merge, in the typing rule for dependent merges there is an extra disjointness condition $A * \Gamma$ to ensure that the new environment has no conflicts. This extra disjointness condition is needed to ensure that reduction is deterministic in E_i .

34:10 Dependent Merges and First-Class Environments

TDOS environment-based semantics. In E_i , an environment-based semantics, expressed by a reduction relation of the form $v \vdash e_1 \hookrightarrow e_2$, is employed to capture the dynamic behavior of expressions. In contrast to more conventional small-step reduction relations, which are typically based on substitution and beta reduction, here v plays the role of the runtime environment and no substitution is needed during reduction. Basically, an environment is stored during evaluation and the expression being evaluated can access it. During the reduction procedure, the environment can be changed *locally*. For example, suppose that the current environment is v , to evaluate a dependent merge e_1, e_2 . The left branch e_1 is evaluated to a value v_1 first. After that, v_1 is merged with v such that e_2 is evaluated under v, v_1 . As a result, e_2 is able to access and fetch v_1 . For instance, the dependent merge

$$\{x = 2\}, \{y = ?.x + ?.x\}$$

is evaluated to $\{x = 2\}, \{y = 4\}$ under \top , since

$$\{y = ?.x + ?.x\}$$

is evaluated to $\{y = 4\}$ under the environment $\top, \{x = 2\}$.

The reduction semantics is based on a *type-directed operational semantics* (TDOS), following the semantics of calculi with the merge operator [21]. As we have seen, type annotations can be used to remove information from values. Thus, unlike many other calculi, the semantics of E_i is type-dependent. That is, types affect the runtime behavior. To deal with such type-dependent semantics based on giving an operational behavior to type annotations we use a TDOS. In the TDOS there is a *casting* relation $v \hookrightarrow_A v'$, where types are used to guide reduction. Since an environment can be selected by a type annotation, casting also acts as a tool for *synthesizing* values in E_i . During the reduction of an annotated query $? : A$ under environment v , casting is triggered, and v' is synthesized as the result. Take the program above as an example, to evaluate $? : \{x : \text{Int}\}$, which is needed in the projection $?.x$, the following cast is triggered:

$$\top, \{x = 2\} \hookrightarrow_{\{x:\text{Int}\}} \{x = 2\}$$

In essence, the cast extracts the value $\{x = 2\}$ matching the type being cast. With this value, we can further build an expression for the right part of the merge.

Abstractions in E_i . In E_i , an abstraction has the form $\{e\}^m$ where m denotes a mode. There are two modes for abstractions: \bullet and \circ . Here we focus on $\{e\}^\bullet$. Compared with a normal lambda abstraction $\lambda x.e$, there is no variable binding in $\{e\}^\bullet$, since values in the environment are *looked up by types* via the query construct instead of by variable names. For example, after $\{?\}^\bullet : \text{Int} \rightarrow \text{Int}$ is applied with integer 1, the input 1 is put in the environment for evaluating $? : \text{Int}$, and then the query construct looks up a value of type Int , which is 1. We require that a well-typed abstraction $\{e\}^\bullet$ has a type annotation. The (slightly simplified) typing rule for abstractions is:

$$\frac{\Gamma * A \quad \Gamma \& A \vdash e \Leftarrow B}{\Gamma \vdash \{e\}^\bullet : A \rightarrow B \Rightarrow A \rightarrow B} \text{TYP-ABS}$$

Similarly to typing normal lambda abstractions, where a typing assumption $x : A$ is added to the typing context, for typing $\{e\}^\bullet$ in E_i , the input type of $\{e\}^\bullet$ is added into the context to type check the body e . For example, $\text{Top} \vdash \{?\}^\bullet : \text{Int} \rightarrow \text{Int} \Rightarrow \text{Int} \rightarrow \text{Int}$ is valid, since under $\text{Top} \& \text{Int}$, $?$ can check against Int . Besides, there is also a disjointness condition in this rule, which ensures that there are no conflicts between the context and the input type.

Ambiguity would happen without such a condition since, if the body e contains a $?$, there would be different answers to the query $?$, as shown in the following example ($\Gamma \vdash e$ is used to denote the situation that the current context for e is Γ):

$$\text{Int} \vdash (\{?\}^\bullet : \text{Int} \rightarrow \text{Int}) 2$$

Suppose that the current environment contains only the value 1, which is of type Int . After the function is applied to 2, both 1 and 2 appear in the environment, and they have the same type Int . If $?$ desires a value of type Int , then there are two candidates, which results in ambiguity. Thus the condition $\Gamma * A$ prevents such programs. On the other hand, the following program is safe in the context Int , since there is only one value, which is 1, having type Int in the environment.

$$\text{Int} \vdash (\{?\}^\bullet : \text{Bool} \rightarrow \text{Int}) \text{true}$$

In general, conventional calculi where variables are involved normally ensure that a typing context is unique, i.e., all variables in it are distinct. In our calculus, disjointness plays a similar role as uniqueness. A function cannot accept expressions that have overlapping types with the current context. For record types, $\{x : \text{Int}\}$ is not disjoint with itself, so the following is not allowed:

$$\{x : \text{Int}\} \vdash (\{?.x\}^\bullet : \{x : \text{Int}\} \rightarrow \text{Int}) \{x = 1\}$$

In contrast, the following expression is well-typed in the context $\{x : \text{Int}\}$, since two record types are disjoint if they have distinct labels:

$$\{x : \text{Int}\} \vdash (\{?.x\}^\bullet : \{y : \text{Int}\} \rightarrow \text{Int}) \{y = 1\}$$

Note that the use of records and distinct label names is how we can model conventional functions that take several arguments of the same type. That is, we can use labels to unambiguously distinguish between arguments of the same type, similarly to the use of distinct variable names in conventional lambda abstractions.

The abstractions in E_i essentially abstract over an interface if we view interfaces as types. The example from Cardelli in Section 2.1 can be encoded in our calculus:

$$\begin{aligned} \{M = \{x = 3\}\} \\ \{N = \{f = \{?.y + ?.x\}^\circ : \{y : \text{Int}\} \rightarrow \text{Int}\}, \{z = (?f) (?x)\}^\bullet : \{x : \text{Int}\} \rightarrow \{f : \text{Int} \rightarrow \text{Int}\} \& \{z : \text{Int}\}\} \end{aligned}$$

Each module is modelled as a record (if the module does not import anything) or a function that returns a record (if the module imports something). A group of related definitions is expressed as a dependent merge of some other records. An interface, such as the interface of N , that contains typing assumption(s) is encoded as input type(s) of an abstraction, and the export list is the output type. Since merges are dependent in E_i , in the second module z is able to call f . With the \circ mode abstraction, standard lambda abstractions can also be encoded (we will discuss this in Section 5). Both modules are typeable separately (in the empty context). Moreover, we can apply N with M , since $(?.N) (?M)$ is typeable in the context containing N and M . Note that such an application is not expressible in Cardelli's work, since modules are not first-class in his setting.

Closures as a special case of boxes. As in usual environment-based semantics, closures are used in E_i to keep lexical environments around. However, given that we have the box construct in E_i , we do not need to invent a separate construct for closures. In fact, closures

34:12 Dependent Merges and First-Class Environments

have the form $v \triangleright \{e\}^\bullet : A \rightarrow B$, which is just a special case of a box. In a box closure, the environment is a value and the expression under the environment is an annotated abstraction. Note that closures are values and the abstraction inside is not evaluated. Instead, when a closure is applied with a value, the value is put in the environment of the closure, and the body of the abstraction is going to be evaluated under the extended environment. Take the following evaluation as an example:

$$\begin{aligned} & (\{\{?\}^\bullet : \text{Int} \rightarrow \text{Bool}\}^\bullet : \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool}) \text{ true } 1 \\ \hookrightarrow & (\top \triangleright \{\{?\}^\bullet : \text{Int} \rightarrow \text{Bool}\}^\bullet : \text{Bool} \rightarrow \text{Int} \rightarrow \text{Bool}) \text{ true } 1 \\ \hookrightarrow^* & (\top, \text{true} \triangleright \{?\}^\bullet : \text{Int} \rightarrow \text{Bool}) 1 \\ \hookrightarrow & \top, \text{true}, 1 \triangleright ? : \text{Bool} \\ \hookrightarrow^* & \text{true} \end{aligned}$$

The abstraction takes a boolean and an integer as input and returns the boolean. At first, it is packed up with the empty environment to form a closure. Then the two values `true` and `1` are merged with the environment to evaluate the body `? : Bool`. With casting, the annotated query is evaluated to `true`.

Encoding λ_i . To demonstrate the expressiveness of E_i we show that it can encode all well-typed programs in the λ_i calculus [32]: an existing calculus with non-dependent merges and without first-class environments. There are two non-obvious obstacles in the encoding. Firstly, unlike E_i , the λ_i calculus is a conventional lambda calculus with conventional lambda abstractions and variables. Our encoding of λ_i shows that queries and abstractions in E_i can encode conventional variables and lambda abstractions. The second obstacle in the encoding is that dependent merges have more disjointness constraints than non-dependent merges. Therefore, it is not clear how some non-dependent merges may be encoded. However, a combination of dependent merges and other constructs in the E_i calculus enables an encoding of all non-dependent merges. Section 5 details the encoding and proves that all typeable programs in λ_i are encodable and typeable in E_i .

3 The E_i Calculus

In this section we present the E_i calculus, which is a calculus with dependent merges and first-class environments. In E_i , type contexts are types, and run-time environments can be assembled, composed, manipulated explicitly, and used to run computations under such environments.

3.1 Syntax

The syntax of E_i is as follows:

Labels	l, x, y, z, \dots
Types and Contexts	$A, B, \Gamma ::= \text{Int} \mid \text{Top} \mid A \rightarrow B \mid A \& B \mid \{l : A\}$
Function modes	$m ::= \bullet \mid \circ$
Expressions	$e ::= ? \mid i \mid \top \mid \{e\}^m \mid e_1 \triangleright e_2 \mid e_1 e_2 \mid e_1, e_2 \mid e : A \mid \{l = e\} \mid e.l$
Values	$v ::= i \mid \top \mid v \triangleright (\{e\}^\bullet : A \rightarrow B) \mid v \triangleright (\{e\}^\circ : \{l : A\} \rightarrow B) \mid \{l = v\} \mid v_1, v_2$

Types and contexts. In E_i there is no syntactic distinction between types and contexts: contexts are types and any type can be a context. In standard calculi typing contexts are lists of typing assumptions of the form $x : A$ that associates variable x with type A . This

particular case is encoded in E_i with a single-field record type $\{x : A\}$. For clarity, we use different meta-variables to denote different uses of types (A, B, C , etc.) and contexts (Γ). Two basic types are included: the integer type Int and the top type Top . Function types and intersection types are created with $A \rightarrow B$ and $A \& B$ respectively. $\{l : A\}$ denotes a record type in which A is the type of the field. Multi-field record types can be desugared to an intersection of single-field record types [32, 40].

Expressions. Meta-variable e ranges over expressions. Expressions include some constructs in standard calculi with a merge operator: integers (i); a canonical top value \top , which can be seen as a merge of zero elements; annotated expressions ($e : A$); application of a term e_1 to term e_2 (denoted by $e_1 \ e_2$); and merge of expressions e_1 and e_2 ($e_1 \ ; \ e_2$). The expression $\{l = e\}$ denotes a single-field record where l is the label and e is its field. Similarly to record types, a multi-field record can be viewed as a merge of single-field records. Projection $e.l$ selects the field from e via the label l .

Besides these standard constructs, there are some novel constructs in our system. Unlike standard calculi, where variables are used to lookup values, we borrow the query construct $?$ from implicit calculi [12] to *synthesize values by types*. However, unlike implicit calculi, in E_i we can completely eliminate the need for variables, since a combination of queries and other constructs can encode traditional uses of variables. Such encoding will be discussed in detail in Section 5. The absence of variables simplifies binding in comparison to other calculi. $\{e\}^m$ stands for *abstractions* in which m is the mode of an abstraction and can be either \bullet or \circ . Abstractions play the same role as lambda abstractions, but they abstract over the input type of the function, instead of abstracting over a variable. The \circ mode denotes a special form of abstraction that is useful to encode lambda abstractions. The term $e_1 \triangleright e_2$ is called a *box*. A box assigns a local environment e_1 for e_2 , and e_2 is not affected by the global context or environment. In other words, boxes allow the computation of e_2 to be performed under the runtime environment resulting from e_1 .

Values. The meta-variable v ranges over values. Values include integers, the canonical \top value, *closures*, merges of values and records in which the field is a value. Closures are a special kind of box, in which the local environment e_1 is a value and e_2 is an annotated abstraction. For closures, the type annotation for $\{e\}^\bullet$ can be any arrow type, whereas the input type of the type annotation for $\{e\}^\circ$ can only be a record type.

3.2 Subtyping and Disjointness

Subtyping. The subtyping rules, shown in Figure 1, are standard for a calculus with intersection types [13], but they include an additional rule S-RCD for subtyping record types. Note that the combination of the subtyping rules for intersection types and record types enables us to express both depth and width subtyping for multi-field record types (which are just encoded as intersections of single-field record types). This extended subtyping relation is reflexive and transitive [22].

Disjointness. Compared to λ_i , disjointness is defined in a slightly different way, inspired by an approach suggested by Rehman et al. [37]. To make two functions or two records mergeable, we define disjointness based on ordinary types whose definition is shown in Figure 1. There are two variants of ordinary types in E_i . The one for defining disjointness contains premises marked in gray. In this variant, ordinary types are inductively defined to be types where the top type and intersection types can never appear (except as input types of functions). With the help of ordinary types, we define disjointness as:

34:14 Dependent Merges and First-Class Environments

$A <: B$			<i>(Subtyping)</i>
$\frac{\text{S-Z}}{\text{Int} <: \text{Int}}$	$\frac{\text{S-TOP}}{A <: \text{Top}}$	$\frac{\text{S-ARR} \quad B_1 <: A_1 \quad A_2 <: B_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2}$	$\frac{\text{S-ANDL} \quad A_1 <: A_3}{A_1 \& A_2 <: A_3}$
$\frac{\text{S-ANDR} \quad A_2 <: A_3}{A_1 \& A_2 <: A_3}$	$\frac{\text{S-AND} \quad A_1 <: A_2 \quad A_1 <: A_3}{A_1 <: A_2 \& A_3}$	$\frac{\text{S-RCD} \quad A <: B}{\{l : A\} <: \{l : B\}}$	
Ordinary A			<i>(Ordinary Types)</i>
$\frac{\text{O-INT}}{\text{Ordinary Int}}$	$\frac{\text{O-ARROW} \quad \text{Ordinary } B}{\text{Ordinary } A \rightarrow B}$	$\frac{\text{O-RCD} \quad \text{Ordinary } B}{\text{Ordinary } \{l : B\}}$	

■ **Figure 1** Subtyping and ordinary types.

► **Definition 1** (Disjointness). $A * B \equiv \neg(\exists C, \text{Ordinary } C \wedge A <: C \wedge B <: C)$

Two types are disjoint if and only if the two types do not share any common ordinary supertype. We have proved that our definition of disjointness is equivalent to the one employed by Huang et. al [22] in their formulation of λ_i . This definition states that atomic values, which can inhabit the two types, cannot have overlapping types. Importantly, our definition allows two arrow types or two record types to be disjoint. For example, $\text{Int} \rightarrow \text{Bool}$ is disjoint with $\text{Int} \rightarrow \text{Char}$ as the two types do not share a common ordinary supertype. Note that there is also an equivalent algorithmic definition of disjointness, which is shown in the appendix. Some of the fundamental properties of disjointness are shown next:

► **Lemma 2** (Disjointness Properties). *Disjointness satisfies:*

1. If $A * B$, then $B * A$.
2. $A * (B \& C)$ if and only if $A * B$ and $A * C$.
3. If $A * (B_1 \rightarrow C)$, then $A * (B_2 \rightarrow C)$.
4. $A * B$ if and only if $\{l : A\} * \{l : B\}$.
5. $C * D$ if and only if $(A \rightarrow C) * (B \rightarrow D)$.
6. If $A <: B$ and $A * C$, then $B * C$.

3.3 Bidirectional Typing

The type system of E_i shown in Figure 2 is bidirectional. There are two modes of typing, where \Rightarrow and \Leftarrow denote the synthesis and checking modes respectively. The notation \Leftrightarrow is a metavariable for typing modes. The meaning of typing judgment $\Gamma \vdash e \Leftrightarrow A$ is standard: under the context Γ (which is a type), expression e can synthesize (with \Rightarrow) or check against (with \Leftarrow) A .

Typing the query construct. Rule TYP-CTX states that $?$ can synthesize the context. With rule TYP-SUB, $?$ checks against any type that is a supertype of the context. In addition, with rule TYP-ANNO, under a context Γ , for any supertype A of Γ , $? : A$ can synthesize A . Since

$$\boxed{\Gamma \vdash e \Leftarrow A} \quad (\text{Bidirectional Typing})$$

$$\begin{array}{c}
\text{TYP-LIT} \quad \text{TYP-CTX} \quad \text{TYP-TOP} \quad \text{TYP-ANNO} \\
\frac{}{\Gamma \vdash i \Rightarrow \text{Int}} \quad \frac{}{\Gamma \vdash ? \Rightarrow \Gamma} \quad \frac{}{\Gamma \vdash \top \Rightarrow \text{Top}} \quad \frac{}{\Gamma \vdash e \Leftarrow A} \\
\frac{}{\Gamma \vdash e : A \Rightarrow A}
\end{array}$$

$$\begin{array}{c}
\text{TYP-ABS} \quad \text{TYP-APP} \\
\frac{\Gamma * A \quad C <: A_m \quad \Gamma \& A \vdash e \Leftarrow B}{\Gamma \vdash \{e\}^m : A \rightarrow B \Rightarrow C \rightarrow B} \quad \frac{\Gamma \vdash e_1 \Rightarrow A \rightarrow B \quad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1 e_2 \Rightarrow B}
\end{array}$$

$$\begin{array}{c}
\text{TYP-BOX} \quad \text{TYP-RCD} \quad \text{TYP-PROJ} \\
\frac{\Gamma \vdash e_1 \Rightarrow \Gamma_1 \quad \Gamma_1 \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1 \triangleright e_2 \Rightarrow A} \quad \frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{l = e\} \Rightarrow \{l : A\}} \quad \frac{\Gamma \vdash e \Rightarrow \{l : A\}}{\Gamma \vdash e.l \Rightarrow A}
\end{array}$$

$$\begin{array}{c}
\text{TYP-SUB} \quad \text{TYP-MERGEV} \\
\frac{\Gamma \vdash e \Rightarrow A \quad A <: B}{\Gamma \vdash e \Leftarrow B} \quad \frac{\Gamma \vdash v_1 \Rightarrow A \quad \Gamma \vdash v_2 \Rightarrow B \quad v_1 \approx v_2}{\Gamma \vdash v_1, v_2 \Rightarrow A \& B}
\end{array}$$

$$\begin{array}{c}
\text{TYP-DMERGE} \quad \text{Type Extraction } A_m \\
\frac{\Gamma \vdash e_1 \Rightarrow A \quad \Gamma \& A \vdash e_2 \Rightarrow B \quad A * \Gamma \quad A * B}{\Gamma \vdash e_1, e_2 \Rightarrow A \& B} \quad \boxed{\begin{array}{l} A_{\bullet} = A \\ \{l : A\}_{\circ} = A \end{array}}
\end{array}$$

■ **Figure 2** Bidirectional type system of E_i . The syntax for the bidirectional modes is defined as $\Leftarrow ::= \Rightarrow \mid \Leftarrow$.

contexts are types in our system, a supertype of a type means a portion of a typing context. By annotating $?$ with a supertype of the context, we can proactively pick the desired type information (or equivalently, hide part of type information) from the context. For example, $\text{Int} \& \text{Bool} \vdash ? : \text{Int} \Rightarrow \text{Int}$ is valid, and allows us to pick Int from a typing context with $\text{Int} \& \text{Bool}$.

$$\begin{array}{c}
\text{TYP-SUB} \quad \frac{\text{Int} \& \text{Bool} \vdash ? \Rightarrow \text{Int} \& \text{Bool} \quad \text{Int} \& \text{Bool} <: \text{Int}}{\text{Int} \& \text{Bool} \vdash ? \Leftarrow \text{Int}} \\
\text{TYP-ANNO} \quad \frac{}{\text{Int} \& \text{Bool} \vdash ? : \text{Int} \Rightarrow \text{Int}}
\end{array}$$

Typing abstractions. Rule TYP-ABS is the typing rule for abstractions. An abstraction can synthesize an arrow type, in which the shape of the input type is determined by the mode. For $\{e\}^{\bullet} : A \rightarrow B$ we simply synthesize the type $A \rightarrow B$. For $\{e\}^{\circ}$ with an annotation, $\{e\}^{\circ}$ is well-typed only if the input type is a record type. Furthermore $\{e\}^{\circ} : \{l : A\} \rightarrow B$ synthesizes $A \rightarrow B$ where A is extracted from $\{l : A\}$. This peculiar treatment of $\{e\}^{\circ} : \{l : A\} \rightarrow B$ is because we wish to be able to model conventional lambda abstractions of the form $\lambda l. e : A \rightarrow B$ faithfully. In conventional lambda abstractions, the labels or variable names are only used *internally*, but they are not reflected on the type. The $\{e\}^{\circ}$ abstractions model this behavior and also hide the label information on the type. While an abstraction with the \bullet mode accepts an expression of a type which is the exact input type of its annotation, a well-typed abstraction with the \circ mode can only have an annotation of form $\{l : A\} \rightarrow B$. The label information for the input type is forgotten for the overall type of the abstraction.

Note also that, for obtaining type preservation, there is a subtyping condition in rule TYP-ABS, similarly to the approach employed by Huang and Oliveira [21]. In an implementation of E_i , this subtyping condition can be omitted and we can let $\{e\}^m : A \rightarrow B$ infer $A_m \rightarrow B$ directly, since the condition is only used in E_i to ensure that closures, which are used during

34:16 Dependent Merges and First-Class Environments

reduction at runtime, are type-preserving. In addition to avoiding ambiguity of the type-based lookups, when we introduce assumptions into the context, we need to ensure that the new assumptions are disjoint to the existing assumptions in the environment. Thus rule TYP-ABS also has a disjointness premise to ensure this.

Typing dependent merges. Rule TYP-DMERGE is the typing rule for merges. Unlike previous work for intersection types and the merge operator [22], the merges are *dependent* in our work. For a specific merge $e_1 \text{ , } e_2$, the right branch e_2 may depend on the left branch e_1 . The typing context for e_2 in the premises is the intersection type $\Gamma \& A$, which means that e_2 is affected by not only the global context Γ but also the synthesized type of e_1 . In this way, e_2 can be constructed with the information of e_1 , as illustrated by the following example:

$$\{z : \text{Int}\} \vdash \{x = 1\} \text{ , } \{y = (? : \{x : \text{Int}\}).x + 1\} \Rightarrow \{x : \text{Int}\} \& \{y : \text{Int}\}$$

The right branch $\{y = (? : \{x : \text{Int}\}).x + 1\}$ makes use of the type information of the left branch, by using $?$ to pick $\{x : \text{Int}\}$ from $\{z : \text{Int}\} \& \{x : \text{Int}\}$. Then it will be able to utilize the value information from $\{x = 1\}$ to evaluate the expression in the right branch of the merge.

There are two disjointness conditions in rule TYP-DMERGE. One is $A * B$, which makes two branches e_1 and e_2 be merged safely without ambiguities as in previous work [22]. However, this condition is not sufficient to prevent all the conflicts between values when the merges are dependent. An additional disjointness condition $A * \Gamma$ is needed to ensure that the synthesized type of the left branch e_1 is disjoint with the context. Without this extra condition, there can be conflicts between e_1 and the current environment. Take the following as an example:

$$(\text{Int} \rightarrow \text{String}) \& \text{Int} \vdash 2 \text{ , } ((? : \text{Int} \rightarrow \text{String}) (? : \text{Int}))$$

The context contains type $\text{Int} \rightarrow \text{String}$ and Int , and the left branch, 2 , has type Int which clashes with the Int that is already in the context. The right branch is an application, which picks a closure and another integer value, say 1 , from the current environment. Suppose that the closure returns the string representation of the input integer. Then $? : \text{Int}$ in the right branch can choose either 1 from the environment or 2 from the left branch, and the merge above can be non-deterministically evaluated to either $2 \text{ , } "1"$ or $2 \text{ , } "2"$. Since we wish to have deterministic evaluation, we prevent such cases with the additional disjointness condition $A * \Gamma$.

Consistency, boxes and closures. Rule TYP-MERGEV is the typing rule for consistent merges. This rule is identical to the rule in previous work using non-dependent merges [21]. Like in previous work, rule TYP-MERGEV is a special run-time typing rule for merges of values and can be omitted in a programming language implementation. If two consistent values are well-typed then it is safe to merge them together. One may wonder why in this rule the context is not extended with A to type-check v_2 . The reason is that values are *closed*, so they cannot depend on the information that is present in the context. During the reduction process, such information has been already filled in into the values. Consistency is defined in terms of casting (whose definition is shown in Figure 3):

► **Definition 3 (Consistency).** *Two values v_1 and v_2 are said to be consistent (written as $v_1 \approx v_2$) if for any type A , the result of casting for the two values is identical.*

$$v_1 \approx v_2 \equiv \forall A, \text{ if } v_1 \hookrightarrow_A v'_1 \text{ and } v_2 \hookrightarrow_A v'_2 \text{ then } v'_1 = v'_2$$

$$\boxed{v \hookrightarrow_A v'} \quad (\text{Casting})$$

$$\begin{array}{c}
\text{CASTING-INT} \quad \text{CASTING-TOP} \quad \text{CASTING-ARROW} \\
\frac{}{i \hookrightarrow_{\text{Int}} i} \quad \frac{}{v \hookrightarrow_{\text{Top}} \top} \quad \frac{\neg \upharpoonright D \upharpoonright \quad C <: A_m \quad B <: D}{v \triangleright (\{e\}^m : A \rightarrow B) \hookrightarrow_{C \rightarrow D} v \triangleright (\{e\}^m : A \rightarrow D)}
\end{array}$$

$$\begin{array}{c}
\text{CASTING-ARROWTL} \quad \text{CASTING-MERGEVL} \\
\frac{\upharpoonright D \upharpoonright \quad C <: A_m \quad B <: D}{v \triangleright (\{e\}^m : A \rightarrow B) \hookrightarrow_{C \rightarrow D} (C \rightarrow D)^\uparrow} \quad \frac{v_1 \hookrightarrow_A v'_1 \quad \text{Ordinary } A}{v_1 \wp v_2 \hookrightarrow_A v'_1}
\end{array}$$

$$\begin{array}{c}
\text{CASTING-MERGEVR} \quad \text{CASTING-AND} \quad \text{CASTING-RCD} \\
\frac{v_2 \hookrightarrow_A v'_2 \quad \text{Ordinary } A}{v_1 \wp v_2 \hookrightarrow_A v'_2} \quad \frac{v \hookrightarrow_A v_1 \quad v \hookrightarrow_B v_2}{v \hookrightarrow_{A \& B} v_1 \wp v_2} \quad \frac{v \hookrightarrow_A v'}{\{l = v\} \hookrightarrow_{\{l:A\}} \{l = v'\}}
\end{array}$$

■ **Figure 3** Casting of E_i .

Given two values, if they have disjoint types, then they are consistent:

► **Lemma 4** (Disjointness implies consistency). *If $A * B$, $\Gamma_1 \vdash v_1 \Rightarrow A$, and $\Gamma_2 \vdash v_2 \Rightarrow B$, then $v_1 \approx v_2$.*

Rule TYP-BOX is the rule for boxes. To make a box $e_1 \triangleright e_2$ well-typed, the global context Γ is replaced for e_2 with type Γ_1 , which is the synthesized type of the local environment e_1 . In other words, the expression e_2 in the box is only affected by the local context. As a special kind of box, closures are closed since the local environment for them is a value and this information is stored for the abstraction. Thus, it is always safe to change the context for closures to any other context. However, we cannot do that for abstractions. For example, if the context for $\{?\}^\bullet : \text{Int} \rightarrow \text{Int}$ is changed from **Top** to **Int**, then the disjointness condition in rule TYP-ABS is broken.

Generally speaking, changing the typing context may introduce more type information such that disjointness does not hold anymore. For example, suppose that the current context is **Int**, which is disjoint with **Char & Bool**. If we replace **Int** with **Int & Bool**, then the new type information **Bool** is introduced in the context and it conflicts with **Char & Bool**. For disjoint values, Lemma 4 ensures that the values are also consistent, so they can be merged together. Therefore, typing two disjoint values does not rely on rule TYP-DMERGE, which restricts the type of the left branch to be disjoint with the context. In fact, another way to describe the closedness of values is to show that the typing context for values can be replaced arbitrarily:

► **Lemma 5** (Value closedness). *If $\Gamma_1 \vdash v \Leftrightarrow A$, then $\Gamma_2 \vdash v \Leftrightarrow A$.*

3.4 Semantics

We now introduce the call-by-value semantics of E_i using an environment-based operational semantics. The semantics employs a type-directed operational semantics (TDOS) [21]. In TDOS, in addition to a reduction relation, there is also a *casting* relation, which is introduced to reduce values based on the type of a given value.

34:18 Dependent Merges and First-Class Environments

$$\begin{array}{c}
 \text{Frames } F ::= [] : A \mid [], e \mid \{l = []\} \mid [], l \mid [], e \mid v [] \mid [], \triangleright e \\
 \\
 \boxed{v \vdash e \hookrightarrow e'} \qquad \text{(Reduction)} \\
 \\
 \begin{array}{ccc}
 \text{STEP-CTX} & \text{STEP-ANNOV} & \text{STEP-MERGER} \\
 \frac{}{v \vdash ? \hookrightarrow v} & \frac{v_1 \hookrightarrow_A v'_1}{v \vdash v_1 : A \hookrightarrow v'_1} & \frac{v, v_1 \vdash e \hookrightarrow e'}{v \vdash v_1, e \hookrightarrow v_1, e'} \\
 \\
 \text{STEP-CLOSURE} & & \text{STEP-BOX} \\
 \frac{}{v \vdash \{e\}^m : A \rightarrow B \hookrightarrow v \triangleright (\{e\}^m : A \rightarrow B)} & & \frac{v_1 \vdash e \hookrightarrow e' \quad \neg \text{Closure}(v_1 \triangleright e)}{v \vdash v_1 \triangleright e \hookrightarrow v_1 \triangleright e'} \\
 \\
 \text{STEP-BOXV} & \text{STEP-PROJV} & \text{STEP-EVAL} \\
 \frac{}{v \vdash v_1 \triangleright v_2 \hookrightarrow v_2} & \frac{}{v \vdash \{l = v_1\}.l \hookrightarrow v_1} & \frac{v \vdash e \hookrightarrow e'}{v \vdash F[e] \hookrightarrow F[e']} \\
 \\
 \text{STEP-BETA} & & \text{Value Construction } A_m^v \\
 \frac{v_1 \hookrightarrow_{A_m} v'_1}{v \vdash (v_2 \triangleright (\{e\}^m : A \rightarrow B)) v_1 \hookrightarrow (v_2, A_m^v) \triangleright (e : B)} & & \boxed{\begin{array}{l} A_\bullet^v = v \\ \{l : A\}_o^v = \{l = v\} \end{array}}
 \end{array}
 \end{array}$$

$$\boxed{v \vdash e \hookrightarrow^* e'} \qquad \text{(Multistep Reduction)}$$

$$\begin{array}{ccc}
 \text{MULTI-REFL} & & \text{MULTI-STEP} \\
 \frac{}{v \vdash e \hookrightarrow^* e} & & \frac{v \vdash e \hookrightarrow e' \quad v \vdash e' \hookrightarrow^* e''}{v \vdash e \hookrightarrow^* e''}
 \end{array}$$

■ **Figure 4** Call-by-value reduction and multistep reduction of E_i .

Casting. The casting relation, shown in Figure 3, is defined on values. The casting relation is essentially the same as the relation in Huang et al.’s work [22]. The only difference is that, instead of having lambda abstractions as values, we now have closures as values. So the rules `CASTING-ARROW` and `CASTING-ARROWTL` change correspondingly to adapt to the new form of values. Rule `CASTING-INT` casts any integer value to itself under type `Int`. Rule `CASTING-TOP` casts any value to a \top under the top type. For merges, rule `CASTING-MERGEVL` and rule `CASTING-MERGEVR` cast one of the two branches under an ordinary type. These two rules can be viewed as value selectors for merges. The definition of ordinary types is the variant without the conditions marked in gray shown in Figure 1. In other words, ordinary types used in casting are those types that are not the top type or intersection types. With rule `CASTING-AND`, a value is cast under two parts of an intersection type respectively, and a merge is returned by combining the two results via the merge operator. Rule `CASTING-RCD` casts a record value under a record type with the same label, and the result is a new record that is constructed from the result of casting the inner value under the inner type of the record type.

A closure $v \triangleright \{e\}^m : A \rightarrow B$ can be cast under an arrow type $C \rightarrow D$ to be a new value. If D is not top-like, then rule `CASTING-ARROW` casts the closure such that the return type is changed to D . Rule `CASTING-ARROWTL` ensures the determinism of casting by casting a closure to be a value generated by the value generator function (A^\dagger) for top-like types. Without this rule, casting a merge of two closures via a top-like type can lead to different results. The definition of top-like types and the value generator are shown in the appendix.

Reduction. Reduction is shown in Figure 4. In the reduction relation $v \vdash e_1 \hookrightarrow e_2$, the environment v is a value. Since environments are involved in reduction, the definition of multi-step reduction is changed accordingly as shown in Figure 4. Briefly speaking, $v \vdash e_1 \hookrightarrow^* e_2$ means that e_1 can be reduced to e_2 by multiple steps under the same environment v , though the environment is possibly changed locally, during single-step reductions.

Synthesizing values by types. Rule STEP-CTX reduces a query $?$ to the current environment. Rule STEP-ANNOV is the rule for annotated values, which triggers casting. In TDOS, casting uses type information from type annotations to guide the reduction to ensure determinism. Moreover, in E_i , casting also allows values to be fetched by types from the environment.

$$\text{MULTI-STEP} \frac{\text{STEP-EVAL} \frac{\text{STEP-CTX} \frac{}{v \vdash ? \hookrightarrow v}}{v \vdash ? : A \hookrightarrow v : A} \quad \text{STEP-ANNOV} \frac{v \hookrightarrow_A v'}{v \vdash v : A \hookrightarrow v'}}{v \vdash ? : A \hookrightarrow^* v'}$$

As shown in the derivation tree above, with $v \hookrightarrow_A v'$, we can conclude that $? : A$ will be evaluated to v' eventually. That is, the answer to a query that is equipped with a specific type, is the result of casting the current environment under that type. For example, suppose that the environment is $1, \text{true}$. Then the answer to the query $? : \text{Int}$ is 1 while the answer to the query $? : \text{Bool}$ is true .

Evaluating dependent merges. Similarly to the reduction strategy in calculi with intersection types and a merge operator, merges are evaluated from left to right in E_i . That is, for a merge e_1, e_2 , the right branch e_2 is evaluated only if the left branch e_1 is a value. However, since merges are dependent in E_i , the evaluation of e_2 relies on e_1 . Specifically, for a merge v_1, e in which v_1 is already a value, rule STEP-MERGER evaluates the right branch e under a new environment v, v_1 such that e can access not only the original environment v but also v_1 . The following is an example of evaluating dependent merges, assuming an initial environment \top :

$$\begin{aligned} & \{x = 1\}, \{y = (? : \{x : \text{Int}\}).x + 1\} \\ \hookrightarrow & \{x = 1\}, \{y = ((\top, \{x = 1\}) : \{x : \text{Int}\}).x + 1\} \\ \hookrightarrow & \{x = 1\}, \{y = (\{x = 1\}).x + 1\} \\ \hookrightarrow & \{x = 1\}, \{y = 1 + 1\} \\ \hookrightarrow & \{x = 1\}, \{y = 2\} \end{aligned}$$

The initial merge is evaluated to $\{x = 1\}, \{y = 2\}$. In every single step of the evaluation above, rule STEP-MERGER is triggered and the right branch $\{y = \dots\}$ is evaluated under $\top, \{x = 1\}$.

Closures and the beta rule. In our call-by-value semantics, when a function, which is not a value, is applied with a value, rule STEP-CLOSURE transforms the function to a closure by assigning the current environment to it. Then rule STEP-BETA reduces the application, where the argument is cast first with the input type of the annotation of the closure. After that, the casting result is merged with the environment in the closure, and this merge becomes the local environment of a box. The body of the box is the body of the abstraction inside the applied closure. Thus, the body of the abstraction will be evaluated further under the new environment, which is a merge carrying the information from both the argument and the environment of the closure.

In rule STEP-BETA, the value $A_m^{v'_1}$ that is added to the environment is different according to the mode of the abstraction. For $v_2 \triangleright (\{e\}^\bullet : A \rightarrow B)$, $A_\bullet = A$ and $A_\bullet^{v'_1} = v'_1$, which is the result of casting v_1 with type A . If the mode is \circ , then the input type for the abstraction can only be a record type, say $\{l : A\}$. Thus for $v_2 \triangleright (\{e\}^\circ : \{l : A\} \rightarrow B)$, $\{l : A\}_\circ = A$ and $\{l : A\}_\circ^{v'_1} = \{l = v'_1\}$. That is, $v_2 \triangleright (\{e\}^\circ : \{l : A\} \rightarrow B)$ can accept a value of type A as input, and the value is given the name l such that it becomes a record during *runtime*. In this way, the body of the abstraction e can use the label to access the information in the record. When the evaluation context is the body of a box, rule STEP-EVAL evaluates the local environment under the global environment until it is a value. After that, rule STEP-BOX evaluates the body of the box under the local environment. A condition is set in rule STEP-BOX to prevent closures from being reduced further. When the body is evaluated to a value, rule STEP-BOXV returns that value.

4 Determinism and Type Soundness

In this section, we show that the operational semantics of E_i is deterministic and type-sound. Unlike previous work on calculi with the merge operator, the typing contexts and the environments appearing in the theorems are generalized to arbitrary ones, since environments are first-class and can be manipulated explicitly in our system.

4.1 Determinism

To obtain the determinism of reduction, the determinism of casting is needed. With the help of consistency, any well-typed value that is cast under the same type results in a unique value.

► **Lemma 6** (Determinism of casting). *If $\Gamma \vdash v \Rightarrow B$, $v \hookrightarrow_A v_1$, and $v \hookrightarrow_A v_2$, then $v_1 = v_2$.*

With determinism of casting, we can prove the following generalized version of determinism, which states that if an expression e is well-typed under the type of the environment v , then the reduction result is the same.

► **Theorem 7** (Generalized determinism). *If $\Gamma \vdash e \Leftrightarrow A$, $\text{Top} \vdash v \Rightarrow \Gamma$, $v \vdash e \hookrightarrow e_1$, and $v \vdash e \hookrightarrow e_2$, then $e_1 = e_2$.*

We cannot prove the standard theorem (where the typing context for e is Top and the environment v is \top) directly. The reason is that the environment is changed in rule STEP-MERGER (from v to $v \circ v_1$) and rule STEP-BOX (from v to v_1). If we prove the standard theorem directly, then the premises in the inductive hypothesis restrict the environment to be \top , which is not strong enough. Therefore, we generalize the theorem. Also note that the typing context for v can be any type in the theorem, since from Lemma 5 we know that the context for a well-typed value can be arbitrary. This fact is important for the proofs of metatheory. When a value is well-typed, we want it also to be well-typed under the context (say Top) appearing in the formalization of the theorem. Consider rule STEP-BOX for example. The environment v_1 in the box is well-typed under the type of v , and it is also well-typed under Top , which meets the condition in the inductive hypothesis.

The standard determinism theorem can then be obtained as a corollary:

► **Corollary 8** (Determinism). *If $\text{Top} \vdash e \Leftrightarrow A$, $\top \vdash e \hookrightarrow e_1$, and $\top \vdash e \hookrightarrow e_2$, then $e_1 = e_2$.*

4.2 Progress and Preservation

For progress and preservation, we need the following properties of casting:

- ▶ **Lemma 9** (Progress of casting). *If $\Gamma \vdash v \Leftrightarrow A$ then there exists v' such that $v \hookrightarrow_A v'$.*
- ▶ **Lemma 10** (Transitivity of casting). *If $v \hookrightarrow_A v_1$ and $v_1 \hookrightarrow_B v_2$ then $v \hookrightarrow_B v_2$.*
- ▶ **Lemma 11** (Consistency after casting). *If $\Gamma \vdash v \Rightarrow C$, $v \hookrightarrow_A v_1$ and $v \hookrightarrow_B v_2$, then $v_1 \approx v_2$.*
- ▶ **Lemma 12** (Preservation of casting). *If $v \hookrightarrow_A v'$ and $\Gamma \vdash v \Rightarrow B$ then $\Gamma \vdash v' \Rightarrow A$.*

These lemmas follow the logic of proving type soundness by Huang and Oliveira [21]. Lemma 9 states that a well-typed value can always be cast with its type. Lemma 10 ensures that casting results in the same value whether a value is cast directly or not. With this property and the determinism of casting, we can prove that the casting results of a value are consistent (Lemma 11), which ensures that casting preserves types (Lemma 12).

Progress and preservation. Similarly to generalized determinism, we have generalized progress and preservation lemmas. Both theorems are proved by induction on the typing judgment.

- ▶ **Theorem 13** (Generalized progress). *If $\Gamma \vdash e \Leftrightarrow A$, then*
 - *e is a value, or*
 - *for any value v , if $\text{Top} \vdash v \Rightarrow \Gamma$, then there exists e' s.t. $v \vdash e \hookrightarrow e'$.*
- ▶ **Theorem 14** (Generalized preservation). *If $\Gamma \vdash e \Leftrightarrow A$, $\text{Top} \vdash v \Rightarrow \Gamma$, and $v \vdash e \hookrightarrow e'$, then $\Gamma \vdash e' \Leftrightarrow A$.*

With the generalized theorems above, the standard progress and preservation theorem can then be obtained as corollaries:

- ▶ **Corollary 15** (Progress). *If $\text{Top} \vdash e \Leftrightarrow A$, then e is a value, or there exists e' s.t. $\top \vdash e \hookrightarrow e'$.*
- ▶ **Corollary 16** (Preservation). *If $\text{Top} \vdash e \Leftrightarrow A$ and $\top \vdash e \hookrightarrow e'$, then $\text{Top} \vdash e' \Leftrightarrow A$.*

Type-safety. Combining generalized progress and preservation, we have generalized type safety where the multistep relation is involved. Basically, this generalized result indicates that under a well-typed environment, a well-typed expression will never get stuck.

- ▶ **Corollary 17** (Generalized type safety). *If $\Gamma \vdash e \Leftrightarrow A$, $\text{Top} \vdash v \Rightarrow \Gamma$, v is a value, and $v \vdash e \hookrightarrow^* e'$, then either e' is a value or there exists e'' s.t. $v \vdash e' \hookrightarrow e''$.*

Thus, the standard type safety is an immediate corollary where the environment is instantiated to be the top value.

- ▶ **Corollary 18** (Type safety). *If $\text{Top} \vdash e \Leftrightarrow A$ and $\top \vdash e \hookrightarrow^* e'$, then either e' is a value or there exists e'' s.t. $\top \vdash e' \hookrightarrow e''$.*

5 Encoding of λ_i

In this section, we show that E_i can encode the type system of the λ_i [32] via a type-directed translation. In other words, every well-typed expression in λ_i can be translated into a well-typed expression in E_i . We do not prove the operational correspondence because of the significant differences between the formulations of the semantics of λ_i and the environment-based semantics of E_i . However, as we discussed in Section 2, the E_i calculus enables first-class environments and dependent merges, which cannot be modelled by λ_i . The translation of λ_i to E_i demonstrates a few different things:

1. **Variables and lambda abstractions are encodable.** The first purpose of this translation is to show that standard variables and lambda abstractions can be fully encoded in E_i . Since λ_i has conventional lambda abstractions, the translation from λ_i to E_i demonstrates that lambdas are encoded in a general way.
2. **Non-dependent merges are encodable.** The second purpose of the translation is to show that non-dependent merges are also encodable. This encoding is not obvious since dependent merges introduce new disjointness restrictions that are not present in calculi such as λ_i . We show that a combination of E_i constructs can express all non-dependent merges without loss of generality.
3. **The E_i calculus subsumes λ_i .** Finally, with the two previous points, we can generally conclude that all typeable programs in λ_i can be encoded in E_i . So E_i is more powerful than λ_i . This is a desirable property since E_i is designed as a potential replacement for λ_i . Therefore, we should be able to express all the programs that are expressible in λ_i .

5.1 Syntax

The definitions of types, expressions, and typing contexts of λ_i are shown as follows:

Types	$A, B ::= \text{Int} \mid \text{Top} \mid A \rightarrow B \mid A \& B$
Expressions	$E ::= x \mid i \mid \top \mid \lambda x. E \mid E_1 E_2 \mid E_1 , , E_2$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : A$

Note that λ_i is a conventional lambda calculus with standard lambda abstractions and a standard context definition. Moreover, in λ_i contexts are *not* types, and environments are not first class.

5.2 Type-Directed Translation of λ_i to E_i

To utilize the information from λ_i contexts to construct expressions of E_i , we need to transform λ_i contexts to E_i contexts which are types. The translation function for contexts is defined as follows.

► **Definition 19** (Context translation). $|\Gamma|$ transforms contexts of λ_i to types of E_i .

$$\begin{aligned} |\cdot| &= \text{Top} \\ |\Gamma, x : A| &= |\Gamma| \& \{x : A\} \end{aligned}$$

Figure 5 shows the typing rules of λ_i with an elaboration into E_i . Four of the rules are straightforward. Rule STYP-LIT simply translates an integer to itself. Similarly, Rule STYP-TOP translates the top value to itself. Rule STYP-SUB produces an expression by adding a type annotation, which is a super type of the type of the expression in the premise. Rule STYP-APP simply combines the two elaborated expressions into an application in E_i .

$$\boxed{\Gamma \vdash E : A \rightsquigarrow e} \quad (\text{Typing with elaboration})$$

$$\begin{array}{c}
\text{STYP-LIT} \qquad \qquad \text{STYP-TOP} \qquad \qquad \text{STYP-VAR} \\
\frac{}{\Gamma \vdash i : \text{Int} \rightsquigarrow i} \qquad \frac{}{\Gamma \vdash \top : \text{Top} \rightsquigarrow \top} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A \rightsquigarrow (? : \{x : A\}).x} \\
\text{STYP-SUB} \qquad \qquad \text{STYP-ABS} \\
\frac{\Gamma \vdash E : A \rightsquigarrow e \quad A <: B}{\Gamma \vdash E : B \rightsquigarrow e : B} \qquad \frac{\Gamma, x : A \vdash E : B \rightsquigarrow e}{\Gamma \vdash \lambda x. E : A \rightarrow B \rightsquigarrow \{e\}^\circ : \{x : A\} \rightarrow B} \\
\text{STYP-APP} \\
\frac{\Gamma \vdash E_1 : A \rightarrow B \rightsquigarrow e_1 \quad \Gamma \vdash E_2 : A \rightsquigarrow e_2}{\Gamma \vdash E_1 E_2 : B \rightsquigarrow e_1 e_2} \\
\text{STYP-MERGE} \\
\frac{\Gamma \vdash E_1 : A \rightsquigarrow e_1 \quad \Gamma \vdash E_2 : B \rightsquigarrow e_2 \quad A * B \quad \text{fresh } x}{\Gamma \vdash E_1, E_2 : A \& B \rightsquigarrow \{x = ?\} \triangleright (? . x \triangleright e_1), ((? : \{x : |\Gamma|\}).x \triangleright e_2)}
\end{array}$$

■ **Figure 5** Type system of λ_i and its type-directed translation into E_i .

Encoding variables. Rule STYP-VAR uses labels to model variables. If a variable x has type A , then $x : A$ must appear in the context. This information from contexts is encoded as a record type $\{x : A\}$ in E_i . Thus, it becomes safe to annotate the query $?$ with $\{x : A\}$. To get the type of x , a record projection is performed to extract the value of type A from $\{x : A\}$.

Encoding lambda abstractions. Similarly, the type information of the bound variable x in a λ_i lambda abstraction is also translated to $\{x : A\}$. For any $\lambda x. E$ of type $A \rightarrow B$, rule STYP-ABS encodes it as $\{e\}^\circ : \{x : A\} \rightarrow B$, which has type $A \rightarrow B$ instead of $\{x : A\} \rightarrow B$. In this way, it can accept values of type A instead of $\{x : A\}$. For example, $\lambda x. x$ with type $\text{Int} \rightarrow \text{Int}$ is translated to $\{(\{x : \text{Int}\}).x\}^\circ : \{x : \text{Int}\} \rightarrow \text{Int}$, which can accept the integer 1 as input in an application.

Encoding non-dependent merges. Merges in λ_i are non-dependent and are encoded in an interesting way in E_i . For dependent merges, the global context should be disjoint with the type of the left branch. To prevent overlapping between $|\Gamma|$ and A , a fresh label x that does not appear in the existing types is picked to create a record $\{x : |\Gamma|\}$ that holds the current environment. This record becomes the context for the merge and is disjoint with A , since x is fresh and consequently A cannot contain a record with a field x . With the box construct, the merge is assigned the local environment $\{x = ?\}$. For the left branch of the merge, projection $? . x$ unwraps the context to take back the original context $|\Gamma|$. Similarly, unwrapping is needed for the right branch. However, since A appears in the typing scope for the right branch in a dependent merge, the annotation $\{x : |\Gamma|\}$ is needed for hiding A . In this way, only $|\Gamma|$ appears in the typing context of e_2 .

Example. In λ_i , the merge $x, \lambda y. y$ can have type $\text{Int} \& (\text{Int} \rightarrow \text{Int})$ in the context $x : \text{Int}$. This expression is translated to the following expression in E_i :

$$\{z = ?\} \triangleright (? . z \triangleright (? : \{x : \text{Int}\}). x), ((? : \{z : \{x : \text{Int}\}\}). z \triangleright \{(? : \{y : \text{Int}\}). y\}^\circ : \{y : \text{Int}\} \rightarrow \text{Int})$$

where z is the fresh label that wraps the environment. This E_i expression infers $\text{Int} \& (\text{Int} \rightarrow \text{Int})$ in the context $\{x : \text{Int}\}$.

Type safety of the translation. The following result shows the type-safety of the translation, and that the type system of λ_i can be translated into E_i without loss of expressivity. Importantly, normal lambda abstractions and non-dependent merges are expressible in E_i .

► **Theorem 20** (Well-typed encoding of λ_i). *If $\Gamma \vdash E : A \rightsquigarrow e$, then $|\Gamma| \vdash e \Rightarrow A$.*

6 Related Work

First-class environments. First-class environments enable environments to be manipulated by programmers. Gelernter et al. [18] invented a programming language called Symmetric Lisp that enriches Lisp with a kind of first-class environment, which can be used to evaluate expressions. They argued using several examples that the first-class environments they defined generalize a variety of constructs including modules, records, closures, and classes. However, the formal semantics of the language is not included in their work. Miller and Rozas [29] also proposed an extension to the Scheme programming language. In their work, environments are created with `make-environment`, and a binary `eval` function is used to perform computations under a first-class environment. Jagannathan [23, 24] defined a dialect of Scheme called Rascal, in which two key operators related to first-class environments are introduced: `reify` that returns the current environment as a data object, and `reflect` which transforms data objects to an environment.

Queinnec and de Roure [36] present a form of first-class environments as an approach to share data objects for the Scheme programming language. Operators on environments, such as composition, importing, and exporting, are supported in their setting. Moreover, the first-class environments they proposed obey the quasi-static discipline [26] such that variables are either static or quasi-static during importing and exporting. Note that our treatment of variable names is similar to the quasi-static scoping approach [26] in some sense. To solve the issue of name capturing, in quasi-static scoping, a free variable has an *internal* name and an *external* name. The external name is for sharing variable bindings and is not α -convertible. The programmer has to resolve it before dereferencing. In our setting, the label x in the abstraction $\{?.x\}^\bullet : \{x : \text{Int}\} \rightarrow \text{Int}$ acts as an external name. In order to avoid ambiguities, the external names in quasi-static scoping must be different in their setting, which is similar to our approach where names are ensured to be different via disjointness.

All the work above is done in a dynamically typed setting. Regarding typed languages, there is little work on first-class environments, which are basically based on explicit substitutions [1]. Sato et al. [44, 45] introduced a simply typed calculus called $\lambda\varepsilon$ with environments as first-class values. In their work, full reduction is supported, and lambda abstractions allow local renaming of bounded variables to fresh names. Sato et al. proved some desirable properties, such as subject reduction, confluence and strong normalizability, for this calculus. First-class environments are called explicit environments in $\lambda\varepsilon$, which are sets of variable-value pairs. Moreover, there is an evaluation operation $e[[a]]$ that evaluates the expression a under an environment e . This construct is similar to the box construct in E_i . However, reification and environment concatenation are not supported in his work. Nishizaki [47]

proposed a similar calculus with first-class environments, in which a construct called *id* is introduced to return the current environment. This construct acts as reification and is similar to our queries, but Nishizaki’s calculus does not support restriction. In E_i queries together with type annotations can retrieve parts of an environment, and model environment restriction. While there is an operator called extension, which can be viewed as a special case of concatenation in Nishizaki’s work, the types do not accumulate. In contrast, environment concatenation in E_i is modelled via dependent merges with type information flowing from left to right. Subtyping is not included in existing type systems with environment types. In contrast, E_i supports subtyping and has a natural notion of subtyping of environments. As a result, it enables more applications. For instance, objects and inheritance can be modelled in E_i [5].

Module systems. Module systems [27] are a key structuring mechanism to build reusable components in modular programming. In ML-style languages, module systems serve as a powerful tool for data abstraction. Generally speaking, a module is a named collection of (dependent) declarations that aim to define an environment. Since dependent merges are supported in E_i , a simple form of modules is allowed by using records and merges in our work. For example, the record $\{M = \{x = 1\}, \{y = ?.x\}\}$ in E_i defines a module named M that contains dependent declarations. Conventionally, ML-style languages are stratified into two parts: a core language, which is associated with ordinary values and types; and a module language consisting of modules and module types (or signatures). In this way, modules are second-class since a module cannot be passed as an argument to a function. In E_i , a simple form of *first-class* modules is enabled via first-class environments. Therefore in our setting, modules can be created and manipulated on the fly. For instance, the above module M encoded as a record can be passed to a function, such that the values bound with x and y could be updated.

There is much work on getting around this stratification to enable first-class modules. One approach is to utilize dependent types. Harper and Mitchell proposed XML calculus [20] which is a dependent type system to formalize modules as Σ and Π types. After that, translucent sums [19] and singleton types [48] were present as extensions and refinement of the XML calculus. On the other hand, Rossberg et al. proposed the F-ing method [42] to encode the ML module system using System F_ω [3] rather than dependent types. Following the F-ing method, 1ML was proposed by Rossberg [41] in which core ML and modules are collapsed into one language. Compared with E_i , the calculi in this kind of work are more expressive due to the use of powerful type systems, where type declarations and abstract types are typically supported. However, expressions, declarations, and modules are separate in the syntax. In contrast, we demonstrate a new approach to enable a simple form of first-class modules via a unified syntax in our work. A variety of entities, including environments, records, declarations, and modules, are simply expressions in E_i .

Implicit calculi. Implicits are a mechanism for implicitly passing arguments based on their types, which are supported in Scala as a generic programming mechanism to reduce boilerplate code. Oliveira et al. [11] investigated the connection between Haskell type classes and Scala implicits. They showed that many extensions of the Haskell type class system can be encoded using implicits. After that, Oliveira et al. [12] synthesized the key ideas of implicits formally in a general core calculus that is called the implicit calculus. The implicit calculus supports a number of source language features that are not supported by type classes. In implicit calculi there are two kinds of contexts and/or environments: there are regular contexts (and environments) tracking variable bindings; and there is also an implicit environment, which tracks values that can be used to provide implicit arguments

automatically. In E_i , we borrow the notion of a query, which enables type-based lookups on implicit environments, from the implicit calculus. While queries are used to query *implicit* environments by type in the implicit calculus, queries in E_i are applied directly to *runtime* environments and there is no distinction between implicit and regular environments.

Rouvoet [43] extended the work of Oliveira et al. and showed that the ambiguous resolution from the implicit calculus is undecidable. Following up on the earlier work on the implicit calculus [12], Schrijvers et al. [46] reformalized the ideas of implicits and presented a coherent and type-safe formal model, which supports first-class overlapping implicits and higher-order rules. Moreover, a more expressive unification-based algorithmic resolution, which is closely related to the idea of propositions as types [50], is described. While a highly complex mechanism is imposed to ensure coherence and the semantics is given by elaboration in their work, in E_i we adopt a TDOS to utilize the type information for guiding reduction and to enable determinism in a natural way. Odersky et al. [31] proposed the SI calculus. The SI calculus generalizes implicit parameters in Scala to implicit function types that have the form of $T? \rightarrow T$, which provides a way to abstract over the contexts consisting of running code. The idea of this generalization was inspired by an early draft of Schrijvers et al.'s work. Unlike the work of Schrijvers et al. and our work, SI lacks unambiguity. Thus a disambiguation scheme is needed in the implementation. While forms of implicit contextual abstraction are offered in the implicit calculi above, a form of contextual abstraction is also supported in E_i . Indeed, since environments are first-class values in E_i , one can easily abstract over the contexts by using abstractions. More recently, Marntirosian et al. [28] added modus ponens to subtyping to make resolution a special case of subtyping and to enable *implicit* first-class environments. Unlike E_i , the runtime environments in their work are still second class.

The merge operator. The merge operator was firstly proposed by Reynolds in the Forsythe language [38] to add the expressiveness for calculi with intersection types. Reynolds' merge operator is quite restrictive and does not allow, for instance, overloaded functions. Since then, several other researchers [8, 15, 32, 33] have removed restrictions and shown more applications of the merge operator. Dunfield [15] presents a powerful calculus with an unrestricted merge operator and an elaboration semantics that can encode various language features. While the elaboration semantics is type-safe, determinism or coherence [39] cannot be ensured. To enable determinism, a disjointness restriction on merges has been proposed in the work of Oliveira et al. [32]. In this work we borrow the idea of merges, intersection types and disjointness from previous work on the merge operator. Unlike previous work, our merges are dependent and E_i has operators to manipulate first-class environments that are not available in earlier calculi with the merge operator. In previous calculi, environments are not first class and the only operators supported on merges are concatenation and restriction.

Staged calculi and modal logic. Staging is a technique to separate the computations of a program, such that abstraction can be realized without loss of efficiency. Davies and Pfenning [14] proposed a type system that captures staged computation based on the intuitionistic variant of the modal logic S4 [35]. The modal necessity operator \Box is introduced, and $\Box A$ represents the type of code that will be evaluated in an upcoming stage. At the term level, expressions of type $\Box A$ have the form $\text{box}(e)$. Corresponding to the modal rule of necessitation, $\text{box}(e)$ has type $\Box A$ if e has type A in the *empty* context. Later, after this work, the box construct is generalized by Nanevski et al. in the work of contextual modal type theory [30]. In this work, the box construct has the form $\text{box}(\Psi.e)$ where Ψ is a context and e can utilize the information in Ψ . The construct $\text{box}(\Psi.e)$ is similar to $e_1 \triangleright e_2$ in

E_i in the sense that the context Ψ shadows the current context. Both constructs capture the dependence of expressions on contexts, in effect modelling data injection. However, since Ψ is a context, e in $\text{box}(\Psi.e)$ can only utilize type information, whereas in $e_1 \triangleright e_2$, e_2 relies on the concrete environment information from the expression e_1 directly. Furthermore, in modal type theory contexts Ψ are defined in the usual way and are not types, nor are first class in the language. In contrast, contexts are types in E_i , and environments are first class values.

Abstract machines. Abstract machines, such as the SECD machine [25], Krivine’s machine, the categorical abstract machine [10], and the CEK machine [16], are state transition systems that serve as a basis for the implementation of functional languages. Typically, a state in abstract machines is a tuple that contains an expression, an environment, and some other entities (such as stack and continuation) for reduction. Similarly, in E_i the semantics is an environment-based semantics, and closures are used to keep environments around during the reduction. However, abstract machines are models for lambda calculus, and thus they are not aimed at providing languages with first-class environments. In contrast, the E_i calculus supports first-class environments and operators that manipulate environments.

7 Conclusion

In this paper, we have presented a statically typed calculus called E_i , that supports the creation, reification, reflection, concatenation and restriction of first-class environments. The E_i calculus borrows disjoint intersection types and a merge operator from the λ_i [32] calculus, but employs them to model environments. In E_i , intersection types are used to model contexts, and disjointness is imposed to model (and generalize) the uniqueness of variables in an environment. However, unlike previous work, merges in E_i are dependent, which enables modelling dependent declarations. From *implicit calculi* [12, 31, 46], E_i borrows queries to synthesize the full current context (at the type level) and the entire current environment (at the term level), and to enable type-based lookups. We prove the determinism and type-soundness of E_i . Furthermore, we show that the type system of λ_i can be encoded by E_i via a type-directed translation. In other words, standard variables, lambda abstractions, and non-dependent merges are all encodable in E_i , enabling the E_i calculus to subsume λ_i . We also study an extension of the calculus with fixpoints. The E_i calculus, as well as the extension, and all the proofs presented in this paper have been formalized using Coq theorem prover.

As for future work, we are interested in extensions with more features. For example, we plan to investigate how to incorporate BCD subtyping [4]. With the merge operator and BCD subtyping, a powerful form of composition called nested composition [6] can be enabled. We would also like to extend the current calculus with polymorphism and show that abstract types can be encoded with the extended calculus. In this setting, since type variables could occur in contexts, we plan to use labels to model type variables, just like what we have done for term variables.

References

- 1 Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, 1989.
- 2 João Alpuim, Bruno C. d. S. Oliveira, and Zhiyuan Shi. Disjoint polymorphism. In *European Symposium on Programming (ESOP)*, 2017.

- 3 Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991. doi:10.1017/s0956796800020025.
- 4 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The journal of symbolic logic*, 48(04):931–940, 1983.
- 5 Xuan Bi and Bruno C. d. S. Oliveira. Typed First-Class Traits. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.
- 6 Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. The Essence of Nested Composition. In *European Conference on Object-Oriented Programming (ECOOP)*, 2018.
- 7 Luca Cardelli. Program fragments, linking, and modularization. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 266–277. ACM Press, 1997. doi:10.1145/263699.263735.
- 8 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Conference on LISP and Functional Programming*, 1992.
- 9 Coq development team. The coq proof assistant. <http://coq.inria.fr/>.
- 10 Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, 1987. doi:10.1016/0167-6423(87)90020-7.
- 11 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360. ACM, 2010. doi:10.1145/1869459.1869489.
- 12 Bruno C. d. S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 35–44. ACM, 2012. doi:10.1145/2254064.2254070.
- 13 Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *International Conference on Functional Programming (ICFP)*, 2000.
- 14 Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001. doi:10.1145/382780.382785.
- 15 Jana Dunfield. Elaborating intersection and union types. *Journal of Functional Programming (JFP)*, 24(2-3):133–165, 2014.
- 16 Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 314–325. ACM Press, 1987. doi:10.1145/41625.41654.
- 17 Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. R melts brains: An ir for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019*, pages 55–66. Association for Computing Machinery, 2019.
- 18 David Gelernter, Suresh Jagannathan, and Thomas London. Environments as first class objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 98–110. ACM Press, 1987. doi:10.1145/41625.41634.
- 19 Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, 1994.
- 20 Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Trans. Program. Lang. Syst.*, 15(2):211–252, 1993. doi:10.1145/169701.169696.

- 21 Xuejing Huang and Bruno C. d. S. Oliveira. A type-directed operational semantics for a calculus with a merge operator. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:32, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.26.
- 22 Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. Taming the merge operator. *Journal of Functional Programming*, 31:e28, 2021. doi:10.1017/S0956796821000186.
- 23 Suresh Jagannathan. Dynamic modules in higher-order languages. In Henri E. Bal, editor, *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, pages 74–87. IEEE Computer Society, 1994. doi:10.1109/ICCL.1994.288391.
- 24 Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Trans. Program. Lang. Syst.*, 16(3):456–492, 1994. doi:10.1145/177492.177578.
- 25 P. J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964. doi:10.1093/comjnl/6.4.308.
- 26 Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 479–492. ACM Press, 1993. doi:10.1145/158511.158706.
- 27 David B. MacQueen. Modules for standard ML. In Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr., editors, *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, pages 198–207. ACM, 1984. doi:10.1145/800055.802036.
- 28 Koar Marntirosian, Tom Schrijvers, Bruno C. d. S. Oliveira, and Georgios Karachalias. Resolution as intersection subtyping via modus ponens. *Proc. ACM Program. Lang.*, 4(OOPSLA):206:1–206:30, 2020. doi:10.1145/3428274.
- 29 James S. Miller and Guillermo Juan Rozas. Free variables and first-class environments. *LISP Symb. Comput.*, 4(2):107–141, 1991.
- 30 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):1–49, 2008.
- 31 Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicity: foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2(POPL), 2017.
- 32 Bruno C. d. S. Oliveira, Zhiyuan Shi, and João Alpuim. Disjoint intersection types. In *International Conference on Functional Programming (ICFP)*, 2016.
- 33 Benjamin C Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, University of Pennsylvania, 1991.
- 34 Garrel Pottinger. A type assignment for the strongly normalizable λ -terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577, 1980.
- 35 Dag Prawitz. *Natural deduction: A proof-theoretical study*. Courier Dover Publications, 2006.
- 36 Christian Queindec and David De Roure. Sharing code through first-class environments. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, pages 251–261. ACM, 1996. doi:10.1145/232627.232653.
- 37 Baber Rehman, Xuejing Huang, Ningning Xie, and Bruno C. d. S. Oliveira. Union Types with Disjoint Switches. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 25:1–25:31, 2022.
- 38 John C Reynolds. Preliminary design of the programming language forsythe. Technical report, Carnegie Mellon University, 1988.
- 39 John C. Reynolds. The coherence of languages with intersection types. In *Lecture Notes in Computer Science (LNCS)*, pages 675–700. Springer Berlin Heidelberg, 1991.

- 40 John C Reynolds. Design of the programming language forsythe. In *ALGOL-like languages*, pages 173–233. Birkhauser Boston Inc., 1997.
- 41 Andreas Rossberg. 1ml - core and modules united. *J. Funct. Program.*, 28:e22, 2018. doi: 10.1017/S0956796818000205.
- 42 Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. *Journal of functional programming*, 24(5):529–607, 2014.
- 43 Arjen Rouvoet. Programs for free: Towards the formalization of implicit resolution in scala. Master’s thesis, TU Delft, 2016.
- 44 Masahiko Sato, Takafumi Sakurai, and Rod M. Burstall. Explicit environments. *Fundam. Informaticae*, 45(1-2):79–115, 2001. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi45-1-2-05>.
- 45 Masahiko Sato, Takafumi Sakurai, and Yuki-yoshi Kameyama. A simply typed context calculus with first-class environments. *J. Funct. Log. Program.*, 2002, 2002. URL: <http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2002/S02-01/JFLP-A02-04.pdf>.
- 46 Tom Schrijvers, Bruno C. d. S. Oliveira, Philip Wadler, and Koar Marntirosian. COCHIS: stable and coherent implicits. *J. Funct. Program.*, 29:e3, 2019. doi:10.1017/S0956796818000242.
- 47 Shin-ya Nishizaki. Simply typed lambda calculus with first-class environments. *Publications of the Research Institute for Mathematical Sciences*, 30(6):1055–1121, 1994.
- 48 Christopher A Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic (TOCL)*, 7(4):676–722, 2006.
- 49 Philip Wadler. The expression problem. *Java-genericity mailing list*, 1998.
- 50 Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, 2015. doi:10.1145/2699407.
- 51 Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. Compositional programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43(3):1–61, 2021.

A Some Relations

A.1 Algorithmic Disjointness

$A \sqcap B$	<i>(COSTs)</i>			
$\frac{\text{COST-INT}}{\text{Int} \sqcap \text{Int}}$	$\frac{\text{COST-ANDL}}{A \sqcap C}$ $\frac{}{A \& B \sqcap C}$	$\frac{\text{COST-ANDR}}{B \sqcap C}$ $\frac{}{A \& B \sqcap C}$	$\frac{\text{COST-RANDL}}{A \sqcap B}$ $\frac{}{A \sqcap B \& C}$	$\frac{\text{COST-RANDR}}{A \sqcap C}$ $\frac{}{A \sqcap B \& C}$
	$\frac{\text{COST-ARR}}{B \sqcap D}$ $\frac{}{A \rightarrow B \sqcap C \rightarrow D}$		$\frac{\text{COST-RCD}}{A \sqcap B}$ $\frac{}{\{l : A\} \sqcap \{l : B\}}$	

Here we define a relation called COSTs (Common Ordinary Super Types), which is used to define algorithmic disjointness as following:

► **Definition 21** (Algorithmic Disjointness). $A *_a B \equiv \neg(A \sqcap B)$

The algorithmic disjointness is equivalent to the specification of disjointness (Definition 1).

► **Theorem 22** (Disjointness Equivalence). $A *_a B$ if and only if $A * B$.

A.2 Top-like Types

$$\boxed{\lceil A \rceil} \quad (\text{Top-like Types})$$

$$\frac{}{\lceil \text{Top} \rceil} \text{TL-TOP} \quad \frac{\lceil A \rceil \quad \lceil B \rceil}{\lceil A \& B \rceil} \text{TL-AND} \quad \frac{\lceil B \rceil}{\lceil A \rightarrow B \rceil} \text{TL-ARR} \quad \frac{\lceil B \rceil}{\lceil \{l : B\} \rceil} \text{TL-RCD}$$

A.3 Value Generator

► **Definition 23** (Value Generator). A^\uparrow generates a value for top-like type A .

$$\begin{aligned} \text{Top}^\uparrow &= \top \\ (A \rightarrow B)^\uparrow &= \top \triangleright (\{B^\uparrow\}^\bullet : A \rightarrow B) \\ (A \& B)^\uparrow &= A^\uparrow, B^\uparrow \\ \{l : A\}^\uparrow &= \{l = A^\uparrow\} \end{aligned}$$

B Fixpoints

In this section, we discuss an extension of E_i with fixpoints.

$$\begin{array}{ll} \text{Expressions} & e ::= \dots \mid \text{fix } A.e \\ \text{Values} & v ::= \dots \mid v \triangleright (\text{fix } A.e : B) \end{array}$$

Syntax and typing. Expressions are extended with fixpoint $\text{fix } A.e$ in which A is the type annotation. For values, closures are extended with boxes containing a fixpoint. Note that for $\text{fix } A.e$ in a closure, an additional type annotation B is required. Rule **TYP-FIX** is the typing rule for fixpoints, which is shown at the top of Figure 6. To make $\text{fix } A.e$ well-typed, the body e needs to be checked under the context extended with A . Similarly to the typing rule for abstractions, there is also a disjointness condition $\Gamma * A$ to prevent ambiguities.

Casting and reduction. The extended casting and reduction rules for fixpoints are shown in Figure 6. Basically, $v \triangleright (\text{fix } A.e : B)$ is cast with a supertype C and the result depends on whether C is top-like or not. If C is not a top-like type, then the casting result is $v \triangleright (\text{fix } A.e : C)$. Otherwise, $v \triangleright (\text{fix } A.e : B)$ is cast to a value generated by the value generator for C . This is similar to the treatment of casting abstractions for ensuring determinism. Note that C is required to be ordinary in rule **CASTING-FIX** and rule **CASTING-FIXTL**. This is to avoid overlapping with rule **CASTING-AND** when C is an intersection type.

For reduction, there are three rules for fixpoints. Rule **STEP-FIX** transforms $\text{fix } A.e$ to a closure by assigning the current environment and giving an additional annotation to it. When $v_2 \triangleright \text{fix } C.e : A \rightarrow B$ is applied to value v_1 , rule **STEP-FIXBETA** “unwinds” the closure in the sense that the closure is put into the environment. In this way, when the application $(e : A \rightarrow B) v_1$ is evaluated, it can access and utilize the closure containing the fixpoint again. Note that the closure put in the environment is $v_2 \triangleright \text{fix } C.e : C$ instead of $v_2 \triangleright \text{fix } C.e : A \rightarrow B$. This is to ensure that the body e of the fixpoint is well-typed under the same context $\Gamma \& C$ for type preservation. Similarly, when a record projection is required, rule **STEP-FIXPROJ** “unwinds” the closure, and evaluates $(e : \{l : B\}).l$ under the environment that contains the fixpoint information.

34:32 Dependent Merges and First-Class Environments

$$\boxed{\Gamma \vdash e \Leftrightarrow A}$$

(Extended Bidirectional Typing)

$$\frac{\text{TYP-FIX} \quad \Gamma * A \quad \Gamma \& A \vdash e \Leftarrow A}{\Gamma \vdash \text{fix } A.e \Rightarrow A}$$

$$\boxed{v \hookrightarrow_A v'}$$

(Extended Casting)

$$\frac{\text{CASTING-FIX} \quad B <: C \quad \neg \lceil C \rceil \quad \text{Ordinary } C}{v \triangleright (\text{fix } A.e : B) \hookrightarrow_C v \triangleright (\text{fix } A.e : C)} \quad \frac{\text{CASTING-FIXTL} \quad B <: C \quad \lceil C \rceil \quad \text{Ordinary } C}{v \triangleright (\text{fix } A.e : B) \hookrightarrow_C C^\uparrow}$$

$$\boxed{v \vdash e \hookrightarrow e'}$$

(Extended Reduction)

$$\frac{\text{STEP-FIX}}{v \vdash \text{fix } A.e \hookrightarrow v \triangleright (\text{fix } A.e : A)}$$

$$\frac{\text{STEP-FIXBETA}}{v \vdash (v_2 \triangleright \text{fix } C.e : A \rightarrow B) v_1 \hookrightarrow (v_2 \text{ , } (v_2 \triangleright \text{fix } C.e : C)) \triangleright (e : A \rightarrow B) v_1}$$

$$\frac{\text{STEP-FIXPROJ}}{v \vdash (v_2 \triangleright \text{fix } A.e : \{l : B\}).l \hookrightarrow (v_2 \text{ , } (v_2 \triangleright \text{fix } A.e : A)) \triangleright (e : \{l : B\}).l}$$

■ **Figure 6** Extended typing, casting, and reduction rules for E_i with fixpoints.

Determinism and type-soundness. The extension with fixpoints retains the properties of determinism and type-soundness. All the metatheory does not require significant changes for this extension and is formalized in the Coq theorem prover.

Synthesis-Aided Crash Consistency for Storage Systems

Jacob Van Geffen ✉ 

University of Washington, Seattle, WA, USA

Xi Wang ✉

University of Washington, Seattle, WA, USA

Amazon Web Services, Seattle, WA, USA

Emina Torlak ✉

University of Washington, Seattle, WA, USA

Amazon Web Services, Seattle, WA, USA

James Bornholt ✉ 

The University of Texas at Austin, TX, USA

Amazon Web Services, Seattle, WA, USA

Abstract

Reliable storage systems must be *crash consistent* – guaranteed to recover to a consistent state after a crash. Crash consistency is non-trivial as it requires maintaining complex invariants about persistent data structures in the presence of caching, reordering, and system failures. Current programming models offer little support for implementing crash consistency, forcing storage system developers to roll their own consistency mechanisms. Bugs in these mechanisms can lead to severe data loss for applications that rely on persistent storage.

This paper presents a new *synthesis-aided* programming model for building crash-consistent storage systems. In this approach, storage systems can assume an *angelic crash-consistency* model, where the underlying storage stack promises to resolve crashes in favor of consistency whenever possible. To realize this model, we introduce a new *labeled writes* interface for developers to identify their writes to disk, and develop a program synthesis tool, DepSynth, that generates *dependency rules* to enforce crash consistency over these labeled writes. We evaluate our model in a case study on a production storage system at Amazon Web Services. We find that DepSynth can automate crash consistency for this complex storage system, with similar results to existing expert-written code, and can automatically identify and correct consistency and performance issues.

2012 ACM Subject Classification Software and its engineering → Search-based software engineering; Computer systems organization → Secondary storage organization

Keywords and phrases program synthesis, crash consistency, file systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.35

Funding This material is based upon work supported by the National Science Foundation under Grant No. 2124044.

1 Introduction

Many applications build on storage systems such as file systems and key-value stores to reliably persist user data even in the face of full-system crashes (e.g., power failures). Guaranteeing this reliability requires the storage system to be *crash consistent*: after a crash, the system should recover to a consistent state without losing previously persisted data. The state of a storage system is consistent if it satisfies the representation invariants of the underlying persistent data structures (e.g., a free data block must not be linked by any file’s inode). Crash consistency is notoriously difficult to get right [34, 22, 35], due to performance optimizations



© Jacob Van Geffen, Xi Wang, Emina Torlak, and James Bornholt;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 35; pp. 35:1–35:26

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in modern software and hardware that can reorder writes to disk or hold pending disk writes in a volatile cache. In normal operation, these optimizations are invisible to the user, but a crash can expose their partial effects, leading to inconsistent states.

A number of general-purpose approaches exist to implement crash consistency, including journaling [23], copy-on-write data structures [24], and soft updates [11]. However, implementing a storage system using these approaches is still challenging for two reasons. First, practical storage systems combine crash consistency techniques with optimizations such as log-bypass writes and transaction batching to improve performance [30]. These optimizations and their interactions are subtle, and have led to severe crash-consistency bugs in well-tested storage systems [17, 7]. Second, developers must implement their system using low-level APIs provided by storage hardware and kernel I/O stacks, which offer no direct support for enforcing consistency properties. Instead they provide only durability primitives such as flushes, and require the developer to roll their own consistency mechanisms on top of them. While prior work offers testing [18, 34] and verification [8, 27] tools for validating crash consistency, these tools do not alleviate the burden of implementing crash-consistent systems.

This paper presents a new synthesis-aided programming model for building crash-consistent storage systems. The programming model consists of three parts: a high-level storage interface based on *labeled writes*; a synthesis engine for turning labeled writes and a desired crash consistency property into a set of *dependency rules* that writes to disk must respect; and a *dependency-aware buffer cache* that enforces the synthesized rules at run time. Together, these three components let developers keep their implementation free of hardcoded optimizations and mechanisms for enforcing consistency. Instead, developers can focus on the key aspects of their storage system – functional correctness, crash consistency, and performance – one at a time. Their development workflow consists of three steps.

First, developers implement their system against a higher-level storage interface by providing *labels* for each write their system makes to disk. Labels provide information about the data structure the write targets and the context for the write (e.g., the transaction it is part of). For example, a simple journaling file system might require two writes to append to the journal: one to append the data block to the tail of the journal (labeled *data*) and one to update a superblock that records a pointer to that tail (labeled *superblock*). This higher-level interface allows the developer to assume a stronger *angelic nondeterminism* model for crashes – the system promises that crash states will *always* satisfy the developer’s crash consistency property if possible – simplifying the implementation effort.

Second, to make their implementation crash consistent even on relaxed storage stacks, the developer uses a new program synthesizer, DepSynth, to automatically generate *dependency rules* that writes to disk must respect. A dependency rule uses labels to define an ordering requirement between two writes: writes with one label must be persisted on disk before corresponding writes with the second label. The DepSynth synthesizer takes three inputs: the storage system implementation, a desired *crash consistency predicate* for disk states of the storage system (i.e., a representation invariant for on-disk data structures), and a collection of small *litmus test* programs [2, 5] that exercise the storage system. Given these inputs, DepSynth searches a space of happens-before graphs to automatically generate a set of dependency rules that guarantee the crash-consistency predicate for every litmus test. Although this approach is example-guided and so only guarantees crash consistency on the supplied tests, the dependency rule language is constrained to make it difficult to overfit to the tests, and so in practice the rules generalize to arbitrary executions of the storage system.

Third, developers run their storage system on top of a *dependency-aware buffer cache* that enforces the synthesized dependency rules. For example, in a journaling file system, the superblock pointer to the tail of the journal must never refer to uninitialized data. DepSynth will synthesize a dependency rule enforcing this consistency predicate by saying that data writes must happen before superblock writes. At run time, the dependency-aware buffer cache enforces this rule by delaying sending writes labeled `superblock` to disk until the corresponding `data` write has persisted. The dependency-aware buffer cache is free to reorder writes in any way to achieve good performance on the underlying hardware (e.g., by scheduling around disk head movement or SSD garbage collection) as long as it respects the dependency rules.

We evaluate the effectiveness and utility of DepSynth in a case study that applies it to ShardStore [4], a production key-value store used by the Amazon S3 object storage service. We show that DepSynth can rapidly synthesize dependency rules for this storage system. By comparing those rules to the key-value store’s existing crash-consistency behavior, we find that DepSynth achieves similar results to rules hand-written by experts, and even corrects an existing crash-consistency issue in the system automatically. We also show that dependency rules synthesized by DepSynth generalize beyond the example litmus tests used for synthesis, and that DepSynth can be used for storage systems beyond key-value stores.

In summary, this paper makes three contributions:

- A new programming model for building storage systems that automates the implementation of crash consistency guarantees;
- DepSynth, a synthesis tool that can infer the dependency rules sufficient for a storage system to be crash consistent; and
- An evaluation showing that DepSynth supports different storage system designs and scales to production-quality systems.

The remainder of this paper is organized as follows. Section 2 gives a walk-through of building a simple storage system with DepSynth. Section 3 defines the DepSynth programming model, including labeled writes and dependency rules. Section 4 describes the DepSynth synthesis algorithm for inferring dependency rules, and Section 5 details DepSynth’s implementation in Rosette. Section 6 evaluates the effectiveness of DepSynth. Section 7 discusses related work, and Section 8 concludes.

2 Overview

This section illustrates the DepSynth development workflow by walking through the implementation of a simple storage system. We show how a developer can build a storage system with labeled writes while assuming a strong crash consistency model, and use DepSynth to automatically make that system crash consistent on real storage stacks.

2.1 Log-structured storage systems

A log-structured storage system persists user data in a sequential log on disk [25]. This design forsakes complex on-disk data structures in favor of one with simple invariants and, as a result, simpler crash consistency requirements. However, although log-structured storage systems are well studied, their precise consistency requirements can be subtle in the face of the caching and reordering optimizations used by the modern storage stack.

Consider implementing a simple key-value store as a log-structured storage system. The on-disk data structure comprises two parts as shown in Figure 2a: a log that stores key-value pairs (with one pair per block), and a superblock that holds pointers to the head and tail of the log. We will assume that single-block writes (`disk.write`) are atomic, that each

```

class KeyValueStore(DepSynth):
    def __init__(self):
        self.superblock = disk.read(0)
        if self.superblock.empty(): # initialize an empty disk
            self.superblock_head, self.superblock_tail = 1, 1
        else:
            self.superblock_head, self.superblock_tail = from_block(superblock)
        self.epoch = 0

    def put(self, key: int, value: int):
        address = self.superblock_tail
        self.superblock_tail += 1

        new_block = to_block(key, value)
        disk.write(address, new_block, ("log", self.epoch))

        new_superblock = to_block(self.superblock_head, self.superblock_tail)
        disk.write(0, new_superblock, ("superblock", self.epoch))

        self.epoch += 1

    def get(self, key: int) -> Optional[int]:
        address = self.superblock_tail - 1
        while address >= self.superblock_head:
            block = disk.read(address)
            current_key, current_value = from_block(block)
            if current_key == key:
                return current_value
            address -= 1
        return None

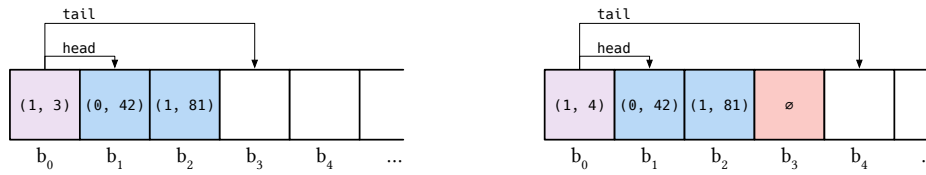
```

■ **Figure 1** Implementation of a simple log-structured key-value store.

key-value pair fits in one block, and that the log does not run out of space. To implement this system, the developer writes `put` and `get` methods that interact with the disk as shown in Figure 1.

Calls to `disk.read` and `disk.write` illustrate our new higher-level storage interface: `disk.read` is unchanged from the usual system call, taking as input an address on the disk to read from; and `disk.write` takes as input an address on the disk to write to, the block data to write to that address, and a third *label* argument. A label is a pair of a string *name* and an integer *epoch*. Labels serve as identities for writes: the name describes the data structure the write targets, while the epoch relates writes across different data structures. This implementation uses the name part of the label to distinguish writes of new log blocks and writes to the superblock,¹ and uses the epoch part as a logical clock that relates the two writes generated by a single `put` call. Labels exist only in memory while a write is in-flight, and are never persisted to disk.

¹ For this system we could distinguish the two data structures without labels – superblock writes are to address 0 while log writes are to non-zero addresses – but in general, storage systems reuse addresses over time and so this mapping is not static.



(a) On-disk layout of a simple log-structured key-value store. Each block holds a (key, value) pair. The first block is a superblock that holds pointers to the head and tail of the log. (b) Possible on-disk state after a crash, leaving the superblock pointing to a range that includes an invalid block.

■ **Figure 2** The on-disk layout of a simple key-value store. Arrows denote pointers and boxes are blocks.

While this implementation is functionally correct, it would not be crash consistent if implemented on a classical storage stack. The issue is with the ordering of log and superblock writes: even though the code suggests that the superblock write comes after the log write, optimizations in the storage stack could reorder the two writes and lead to a crash state where the superblock is updated but its corresponding new log block is not, as Figure 2b shows. This would leave the `superblock_tail` pointer referring to an uninitialized disk block. What we need for consistency is a way to preclude this reordering. One solution in the DepSynth programming model would be for the developer to manually implement a *dependency rule* that prevents this reordering:

```
def __init__(self):
    self.rule("superblock", "log", eq)
```

A dependency rule `rule("a", "b", eq)` specifies an ordering constraint: a write labeled with name "a" must not be sent to disk until after a write labeled with name "b". We say that such a rule means write "a" *depends on* write "b", or equivalently that write "b" must *happen before* write "a". The third argument to `rule` is an *epoch predicate* that scopes the rule using the epoch in each label. Here, the `eq` predicate restricts the rule to only apply to pairs of writes whose labels have equal epochs. This rule means that superblock updates cannot be persisted on disk until a log block write with the same epoch is persisted first, ruling out the reordering behavior that could make the log inconsistent.

2.2 Dependency rule synthesis

While the developer could specify the above dependency rule manually, our programming model does not require them to, and distilling the correct set of rules for a complex storage system is difficult to do by hand. The challenge is a semantic gap: the developer's desired high-level consistency property is about the on-disk data structure as a whole, but the implementation of consistency can only refer to individual block-sized writes. We bridge this gap with DepSynth, a program synthesis tool that can *automatically infer* the dependency rules sufficient to make a storage system crash consistent.

DepSynth takes three inputs. First, it takes as input the implementation of the storage system. Second, it takes as input a crash consistency predicate, written as an executable checker over a disk state. The crash consistency predicate defines the property that should be true of *every* state of the disk, including after crashes. For our log-structured key-value store, our desired consistency property is that the `superblock_tail` pointer never gets ahead of the blocks that have been written to the log. We can implement this property by checking that all blocks in the log are valid log blocks (we omit an implementation of `valid` for brevity, but it could validate a checksum of the block):

35:6 Synthesis-Aided Crash Consistency for Storage Systems

```
def consistent(self) -> bool:
    ret = True
    for address in range(self.superblock_head, self.superblock_tail):
        block = disk.read(address)
        ret = ret and valid(block)
    return ret
```

Finally, DepSynth takes as input a collection of *litmus tests*, small programs that exercise the storage system. Litmus tests are widely used to communicate the semantics of memory consistency models [2, 32], and have also been used to communicate crash consistency models [5]. A DepSynth litmus test comprises two executable programs *initial* and *main*. Both programs take as input a reference to the storage system. The *initial* program sets up some initial state in the system, and cannot crash. The *main* program manipulates the system state, and can crash at any point. For example, this is a simple litmus test that starts from a single log entry and appends two more:

```
class SingleEntry_TwoAppend(LitmusTest):
    def initial(self, store: KeyValueStore):
        store.put(0, 42)

    def main(self, store: KeyValueStore):
        store.put(1, 81)
        store.put(2, 37)
```

As with previous work on memory consistency models [2, 6], the developer can draw litmus tests from a number of sources: they may be hand-written by the developer, drawn from a common set of tests for important properties, generated automatically by a fuzzer or program enumerator, or intelligently generated by analyzing the on-disk data structures used by the storage system [1].

Given these three inputs, DepSynth automatically synthesizes a set of dependency rules that suffice to guarantee the crash-consistency predicate holds on all crash states generated by all litmus tests. For our example log-structured key-value store, DepSynth synthesizes two dependency rules:

```
def __init__(self):
    self.rule("superblock", "log", eq)
    self.rule("superblock", "superblock", gt)
```

The first rule is the same rule we hand-wrote earlier. The second rule fixes a subtle crash-consistency bug in our hand-written implementation: while the first rule ensures consistency for a *single* put operation, it still allows `superblock_tail` to get ahead of the log if writes from *multiple* puts are reordered with each other (for example, reordering writes from the first and second puts in the litmus test above). The second rule prevents this reordering using the `gt` epoch predicate, which specifies that a superblock write with epoch i cannot be persisted to disk until all superblock writes with lower epochs $j < i$ are persisted first. The combination of these rules precludes the problematic reordering and guarantees that the superblock always refers to a valid *range* of log blocks, rather than only requiring the block at `superblock_tail` to be valid.

3 Reasoning About Crash Consistency

The DepSynth workflow includes a new high-level interface for building storage systems and a synthesis tool for automatically making those systems crash consistent. This section describes the high-level interface, including labeled writes and dependency rules, and presents a logical encoding for reasoning about crashes of systems that use this interface. Section 4 then presents the DepSynth synthesis algorithm for inferring sufficient dependency rules to make a storage system crash consistent.

3.1 Disk Model and Dependency Rules

In the DepSynth programming model, storage systems run on top of a disk model d that provides two operations:

- $d.\text{write}(a, v, l)$: write a data block v to disk address a with label l
- $d.\text{read}(a)$: read a data block at disk address a

We assume that single-block write operations are atomic, as in previous work [27, 8]. These interfaces are similar to the standard POSIX `pwrite` and `pread` APIs, except that the `write` operation additionally takes as input a *label* for the write. A label $l = \langle n, t \rangle$ is a pair of a *name* string n and an *epoch* integer t . Labels allow the developer to provide identities for each write their system performs, which dependency rules (described below) can inspect to enforce ordering requirements. Although the two components of a label together identify a write, developers use them for separate purposes: the name indicates which on-disk data structure the write targets, while the epoch associates related writes with different names. Names are strings but are not interpreted by our workflow other than to check equality between them. Epochs are integers that dependency rules use as logical clocks to impose orderings on related writes.

3.1.1 Dependency rules

DepSynth synthesizes declarative *dependency rules* to enforce consistency requirements for a storage system that uses labeled writes.

► **Definition 1** (Dependency rule). *A dependency rule $n_1 \rightsquigarrow_p n_2$ comprises two names n_1 and n_2 and an epoch predicate $p(t_1, t_2)$ over pairs of epochs. Given two labels $l_a = \langle n_a, t_a \rangle$ and $l_b = \langle n_b, t_b \rangle$, we say that a dependency rule $n_1 \rightsquigarrow_p n_2$ matches l_a and l_b if $n_a = n_1$, $n_b = n_2$, and $p(t_a, t_b)$ is true.*

Dependency rules define ordering requirements over all writes with labels that match them, and the dependency-aware buffer cache enforces these rules at run time. More precisely, the dependency-aware buffer cache enforces *dependency safety* for all writes it sends to disk:

► **Definition 2** (Dependency safety). *A dependency-aware buffer cache maintains dependency safety for a set of dependency rules R if, whenever a storage system issues two writes $d.\text{write}(a_1, s_1, l_1)$ and $d.\text{write}(a_2, s_2, l_2)$, and a rule $n_a \rightsquigarrow_p n_b \in R$ matches l_1 and l_2 , then the cache ensures the write to a_1 does not persist until the write to a_2 is persisted on disk.*

In other words, all crash states of the disk that include the effect of the first write must also include the effect of the second write. Section 3.3 will specify dependency safety more formally by defining the crash behavior of a disk in first-order logic.

The epoch predicate of a dependency rule reduces the scope of the rule to only apply to some writes labeled with the relevant names. Given two labels $l_1 = \langle n_1, t_1 \rangle$ and $l_2 = \langle n_2, t_2 \rangle$, a dependency rule $n_1 \rightsquigarrow_p n_2$ can use one of three epoch predicates: $=$, $>$, and $<$, which restrict

the rule to apply only when $t_1 = t_2$, $t_1 > t_2$, and $t_1 < t_2$, respectively. These variations allow dependency rules to specify ordering requirements over unbounded executions of the storage system without adding unnecessary dependencies between *all* operations with certain names.

Together, the name and epoch components of labels allow dependency rules to define a variety of important consistency requirements, depending on how the developer chooses to label their writes. For example, if all writes generated by a related operation (e.g., a top-level API operation like `put` in a key-value store) share the same epoch t , then rules using the $=$ epoch predicate can impose consistency requirements on individual operations, such as providing transactional semantics. As another example, rules using the $>$ epoch predicate can be used as barriers for all previous writes, and so can help to implement operations like garbage collection that manipulate an entire data structure.

3.1.2 Dependency-aware buffer cache

At run time, storage systems implemented with the DepSynth programming model execute on top of a *dependency-aware buffer cache*. This buffer cache is configured with a set of dependency rules at initialization time, and enforces those rules on all writes executed by the storage system.

The dependency-aware buffer cache is inspired by previous higher-level storage APIs such as those used by Featherstitch [10] and ShardStore [4], which also provide interfaces for specifying ordering requirements for writes. Both of these interfaces are imperative: they require the developer to manually construct a dependency graph for each write they execute, and so closely intertwine the ordering requirements with the implementation, as constructing these graphs requires sharing graph nodes (*patchgroups* in Featherstitch and *dependencies* in ShardStore) across threads and operations. In contrast, the dependency-aware buffer cache interface is declarative: the dependency rules are configured once, and then automatically applied to all relevant writes without requiring the developer to manually construct graphs or invoke consistency primitives like `fsync`.

The implementation details of the dependency-aware buffer cache are outside the scope of this paper and follow the examples of Featherstitch and ShardStore. An implementation could use a variety of consistency and durability primitives provided by disks, including force-unit-access writes, cache flush commands, or ordering barriers. We trust the correctness of the dependency-aware buffer cache, and specifically we assume it enforces dependency safety (Definition 2).

3.2 Storage Systems and Litmus Tests

To apply DepSynth, developers provide three inputs: a storage system implementation, a collection of litmus tests that exercise the storage system, and a crash consistency predicate for the system.

3.2.1 Storage system implementations

Developers implement a storage system for DepSynth by defining a collection of API operations \mathcal{O} and an implementation function for each operation:

► **Definition 3** (Storage system implementation). *A storage system implementation $\mathcal{O} = \{O_a, O_b, \dots\}$ is a set of API operations O_i and, for each O_i , an implementation function $I_{O_i}(d, \mathbf{x})$ that takes as input a disk state d and a vector of other inputs \mathbf{x} and issues write operations to mutate disk d .*

DepSynth requires implementation functions to support being symbolically evaluated with respect to a symbolic disk state d . In this paper, we use Rosette [28] as our symbolic evaluator; this requires implementation functions to be written in Racket and allows them to be automatically lifted to support the necessary symbolic evaluation, so long as their executions are deterministic and bounded.

We say that a *program* P is a sequence of calls $[O_1(x_1), \dots, O_n(x_n)]$ to API operations $O_i \in \mathcal{O}$. Given a program P , we write $Evaluate_{\mathcal{O}}(P)$ for the function that symbolically evaluates each $I_{O_i}(d, x_i)$ in turn, starting from a symbolic disk d , and returns a *trace* of labeled write operations $[w_1, \dots, w_n]$ that the program performed. The trace does not need to include read operations as they cannot participate in ordering requirements.

3.2.2 Litmus tests

DepSynth synthesizes dependency rules from a set of example *litmus tests*, which are small programs that exercise the storage system and demonstrate its desired consistency behavior. A litmus test $T = \langle P_{initial}, P_{main} \rangle$ is a pair of programs that each invoke operations of the storage system. The initial program $P_{initial}$ sets up an initial state of the storage system by, for example, prepopulating the disk with files or objects. It will be executed starting from an empty disk, and cannot crash. The main program P_{main} then tests the behavior of the storage system starting from that initial state. DepSynth will exercise all possible crash states of the main program.

Litmus tests are widely used to communicate the semantics of memory consistency models to developers [2, 32], and have also been used to communicate crash consistency [5] and to search for crash consistency bugs in storage systems [18]. DepSynth is agnostic to the source of the litmus tests it uses so long as they fit the definition of a program (i.e., are straight-line and deterministic).

3.2.3 Crash consistency predicates

To define crash consistency for their system, developers also provide a *crash-consistency predicate* $Consistent(d)$ that takes a disk state d and returns whether the disk state should be considered consistent. The crash-consistency predicate should include representation invariants for the storage system’s on-disk data structures. For example, a file system like ext2 might require that all block pointers in inodes refer to blocks that are allocated (i.e., no dangling pointers). These properties correspond to those that can be checked by an `fsck`-like checker [12]. The crash-consistency predicate can also include stronger properties such as checking the atomic-replace-via-rename property for POSIX file systems [5, 22].

3.3 Reasoning About Crashes

To reason about the crash behaviors of a storage system, we encode the semantics of dependency rules and litmus tests in first-order logic based on existing work on storage verification [27]. We first encode the behavior of a single write operation, and then extend that encoding to executions of entire programs.

3.3.1 Write operations

We model the behavior of a disk write operation as a transition function $f_{\text{write}}(d, a, v, s)$, that takes four inputs: the current disk state d , the disk address a to write to, the new block value v to write, and a *crash flag* s , a boolean that is used to encode the effect of a crash on the resulting disk state. Given these inputs, f_{write} returns the resulting disk state after applying the operation. The effect of a write operation is visible on the disk only if s is true:

$$f_{\text{write}}(d, a, v, s) = d[a \mapsto \text{if } s \text{ then } v \text{ else } d(a)].$$

3.3.2 Program executions

Given the trace of write operations $[w_1, \dots, w_n] = \text{Evaluate}_{\mathcal{O}}(P)$ executed by a program P against storage system \mathcal{O} , and for each write its corresponding crash flag s_i , we can define the final disk state of the program by just applying the transition function in sequence:

$$\begin{aligned} & \text{Run}([\text{write}(a_1, v_1, l_1), w_2, \dots, w_n], [s_1, \dots, s_n], d) \\ &= \text{Run}([w_2, \dots, w_n], [s_2, \dots, s_n], f_{\text{write}}(d, a_1, v_1, s_1)) \\ & \quad \text{Run}([], [], d) = d \end{aligned}$$

We call the vector $\mathbf{s} = [s_1, \dots, s_n]$ of crash flags for each operation in the trace a *crash schedule*.

Not all crash schedules are possible. At run time, the dependency-aware buffer cache constrains the set of *valid crash schedules* by applying the dependency rules it is configured with:

► **Definition 4** (Valid crash schedule). *Let $[w_1, \dots, w_n] = \text{Evaluate}_{\mathcal{O}}(P)$ be the trace of operations executed by a program P on storage system \mathcal{O} , R be a set of dependency rules, and $\mathbf{s} = [s_1, \dots, s_n]$ the crash schedule for the trace. The crash schedule \mathbf{s} is valid for the program P and set of rules R , written $\text{Valid}_R(\mathbf{s}, P)$, if for all operations $w_i = \text{write}(a_i, v_i, l_i)$ and $w_j = \text{write}(a_j, v_j, l_j)$, whenever there exists a rule $n_a \rightsquigarrow_p n_b \in R$ that matches l_i and l_j , then $s_i \rightarrow s_j$.*

This definition is a logical encoding of dependency safety (Definition 2): if $s_i \rightarrow s_j$, then write w_j is guaranteed to be persisted on disk whenever write w_i is.

Finally, we can define crash consistency for a litmus test $T = \langle P_{\text{initial}}, P_{\text{main}} \rangle$ as a function of a set of dependency rules R :

► **Definition 5** (Single-test crash consistency). *Let $T = \langle P_{\text{initial}}, P_{\text{main}} \rangle$ be a litmus test. Let $d_{\text{initial}} = \text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\text{initial}}), \top, d_0)$ be the disk state reached by running the program P_{initial} against storage system \mathcal{O} on the all-true (i.e., crash-free) crash schedule \top starting from the empty disk d_0 . A set of dependency rules R makes T crash consistent if, for all crash schedules \mathbf{s} such that $\text{Valid}_R(\mathbf{s}, P_{\text{main}})$ is true, $\text{Consistent}(\text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\text{main}}), \mathbf{s}, d_{\text{initial}}))$ holds.*

► **Example 6.** Consider the `SingleEntry_TwoAppend` litmus test from Section 2. Interpreting the initial and main programs gives two traces:

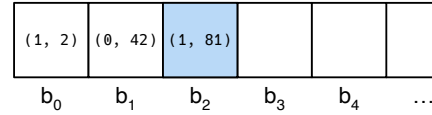
$$\begin{aligned} \text{Interpret}(P_{\text{initial}}) &= [\text{write}(1, \text{to_block}((0, 42)), \langle \text{log}, 0 \rangle), \\ &\quad \text{write}(0, \text{to_block}((1, 2)), \langle \text{superblock}, 0 \rangle)] \\ \text{Interpret}(P_{\text{main}}) &= [\text{write}(2, \text{to_block}((1, 81)), \langle \text{log}, 1 \rangle), \\ &\quad \text{write}(0, \text{to_block}((1, 3)), \langle \text{superblock}, 1 \rangle), \\ &\quad \text{write}(3, \text{to_block}((2, 37)), \langle \text{log}, 2 \rangle), \\ &\quad \text{write}(0, \text{to_block}((1, 4)), \langle \text{superblock}, 2 \rangle)] \end{aligned}$$

Let $\mathbf{s} = [s_1, s_2, s_3, s_4]$ be a crash schedule for P_{main} . Applying the two synthesized rules from Section 2 restricts the valid crash schedules (Definition 4):

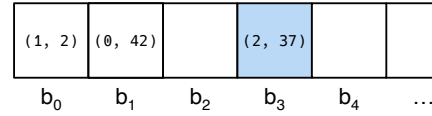
- $\text{superblock} \rightsquigarrow_{=} \text{log}$ requires $s_2 \rightarrow s_1$ and $s_4 \rightarrow s_3$.
- $\text{superblock} \rightsquigarrow_{>} \text{superblock}$ requires $s_4 \rightarrow s_2$.

Combined, these constraints yield seven valid crash schedules. Besides the two trivial crash schedules $\mathbf{s} = \top$ and $\mathbf{s} = \perp$, the other five crash schedules yield five distinct disk states:

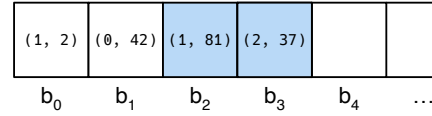
- (1) $[s_1 = \top, s_2 = \perp, s_3 = \perp, s_4 = \perp]$
(only the first log block is on disk)



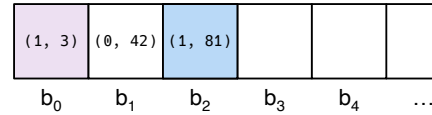
- (2) $[s_1 = \perp, s_2 = \perp, s_3 = \top, s_4 = \perp]$
(only the second log block is on disk)



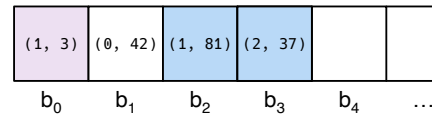
- (3) $[s_1 = \top, s_2 = \perp, s_3 = \top, s_4 = \perp]$
(both log blocks are on disk)



- (4) $[s_1 = \top, s_2 = \top, s_3 = \perp, s_4 = \perp]$
(the first log block and first superblock write are on disk)



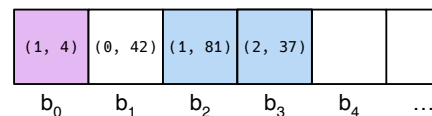
- (5) $[s_1 = \top, s_2 = \top, s_3 = \top, s_4 = \perp]$
(the first log block, first superblock write, and second log block are on disk)



Each of these states satisfies the key-value store’s crash-consistency predicate $\text{Consistent}(d)$ defined in Section 2, as in each case the superblock’s `head` and `tail` pointers refer only to log blocks that are also on disk. Some states result in data loss after the crash – for example, neither key can be retrieved from crash state 1 above, as the superblock is empty – but these states are still consistent (i.e., they satisfy the log’s representation invariant). This set of two rules therefore makes the `SingleEntry_TwoAppend` litmus test crash consistent according to Definition 5.

If the second rule $\text{superblock} \rightsquigarrow_{>} \text{superblock}$ was excluded, the rule set with one remaining rule allows 2 additional crash states:

- (6) $[s_1 = \top, s_2 = \perp, s_3 = \top, s_4 = \top]$
(the first log block, second log block, and second superblock write are on disk)



$[s_1 = \perp, s_2 = \perp, s_3 = \top, s_4 = \top]$
 (7) (the second log block and second superblock write are on disk)

State 6 satisfies the crash-consistency predicate despite losing the first superblock write, as the second superblock write already contains the effects of the first one. However, state 7 violates the crash-consistency predicate: the first log block is invalid, but is included in the range between the superblock’s `head` and `tail` pointers. The set containing only the first rule therefore does not make `SingleEntry_TwoAppend` crash consistent.

4 Dependency Rule Synthesis

This section describes the `DepSynth` synthesis algorithm, which automatically generates a set of dependency rules that are sufficient to guarantee crash consistency for a set of litmus tests. It formalizes the dependency rule synthesis problem, gives an overview of `DepSynth`’s approach to synthesizing dependency rules, and then presents the core `DepSynth` algorithm (Figure 3).

4.1 Problem Statement

`DepSynth` solves the problem of finding a *single* set of dependency rules R that makes every litmus test T in a set of tests \mathcal{T} crash consistent (Definition 5). While Definition 5 suffices to find a set of rules R that guarantees crash consistency, it does not rule out *cyclic* solutions that cannot be executed on real hardware. For example, consider a program P where $Evaluate_{\mathcal{O}}(P) = [\text{write}(a_1, v_1, \langle n_1, t_1 \rangle), \text{write}(a_2, v_2, \langle n_2, t_2 \rangle)]$. The set of rules $R = \{n_1 \rightsquigarrow_{=} n_2, n_2 \rightsquigarrow_{=} n_1\}$ makes P crash consistent. These two rules do not admit any valid crash schedules other than the trivial $s = \top$ and $s = \perp$ schedules, as Definition 4 forces $s_1 = s_2$. In effect, crash consistency for P requires both writes to happen “at the same time”. But on real disks the level of write atomicity is only a single data block, so there is no way for both writes to happen at the same time. To rule out cyclic solutions, we follow the example of happens-before graphs [14] from distributed systems and memory consistency, and require the set of synthesized dependency rules R to be *acyclic*.

4.2 The `DepSynth` Algorithm

The `DEPSYNTH` algorithm (Figure 3) takes as input a storage system implementation \mathcal{O} , a set of litmus tests \mathcal{T} , and a crash-consistency predicate *Consistent*. Given these inputs, it synthesizes a set of dependency rules that is acyclic and sufficient to make all tests \mathcal{T} crash consistent.

`DEPSYNTH` does not try to generate a sufficient set of dependency rules for all tests in \mathcal{T} at once, since this would require a prohibitively expensive search over large happens-before graphs. Instead, it works incrementally: at each iteration of its top-level loop, `DEPSYNTH` chooses a single test T that is not made crash consistent by the current candidate set of dependency rules (line 4 in Figure 3), invokes the procedure `RULESFORTTEST` (Section 4.3) to synthesize dependency rules that make T crash consistent, and adds the new rules to the candidate set (line 11). Working incrementally reduces the number of litmus tests for which `DEPSYNTH` needs to synthesize rules; for example, in Section 6.1 we show that only 10 of 16,250 tests were passed to `RULESFORTTEST` to synthesize a sufficient set of dependency rules for a production key-value store. This reduction relieves developers from being selective about the set of litmus tests they supply to `DepSynth`, and makes it possible to, for example, use the output of a fuzzer or random test generator as input.

```

1 function DEPSYNTH( $\mathcal{O}$ ,  $\mathcal{T}$ , Consistent)
2    $R \leftarrow \{\}$ 
3   loop
4      $T \leftarrow \text{NEXTTEST}(\mathcal{T}, R, \mathcal{O}, \textit{Consistent})$ 
5     if  $T = \perp$  then  $\triangleright R$  makes all tests in  $\mathcal{T}$  crash consistent
6       return  $R$ 
7      $\mathcal{T} \leftarrow \mathcal{T} \setminus T$ 
8      $R' \leftarrow \text{RULESFORTEST}(T, \mathcal{O}, \textit{Consistent})$ 
9     if  $R' = \perp$  then  $\triangleright$  No rules can make  $T$  crash consistent
10      return UNSAT
11      $R \leftarrow R \cup R'$ 
12     if  $\neg \text{ACYCLIC}(R)$  then  $\triangleright$  Fail if new rules create a cycle in the rule set
13       return UNKNOWN
14 function NEXTTEST( $\mathcal{T}$ ,  $R$ ,  $\mathcal{O}$ , Consistent)
15   for  $T \in \mathcal{T}$  do
16     if  $\neg \text{CRASHCONSISTENT}(T, R, \mathcal{O}, \textit{Consistent})$  then
17       return  $T$ 
18   return  $\perp$ 
 $\triangleright$  Check Def. 5 with an SMT solver
19 function CRASHCONSISTENT( $T = \langle P_{\textit{initial}}, P_{\textit{main}} \rangle$ ,  $R$ ,  $\mathcal{O}$ , Consistent)
20    $d_{\textit{initial}} \leftarrow \text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\textit{initial}}), \top, d_0)$ 
21   return  $\forall s. \text{Valid}_R(s, P_{\textit{main}}) \Rightarrow \textit{Consistent}(\text{Run}(\text{Evaluate}_{\mathcal{O}}(P_{\textit{main}}), s, d_{\textit{initial}}))$ 

```

■ **Figure 3** The DepSynth algorithm takes as input a storage system implementation \mathcal{O} , a set of litmus tests \mathcal{T} , and a crash-consistency predicate *Consistent*, and returns an acyclic set of dependency rules that make all tests in \mathcal{T} crash consistent (Definition 5). The search synthesizes dependency rules for one litmus test at a time. If the rules generated for two or more tests result in a cycle, this algorithm fails; Section 4.4 discusses an extension for continuing the search for an acyclic solution.

However, because the rules for each test are generated independently, it is possible for the union of the generated rules to contain a cycle – even if the rules for each individual test do not – and so be an invalid solution (Section 4.1). The algorithm in Figure 3 returns UNKNOWN if such a cycle is found. We have not seen this failure mode occur for the storage systems we evaluated (Section 6), but it is possible in principle. In Section 4.4, we explain how to extend DEPSYNTH to recover from cycles by generalizing RULESFORTEST to synthesize rules for multiple tests at once.

DEPSYNTH delegates checking for crash consistency to the procedure CRASHCONSISTENT (line 19), which takes as input a single litmus test and a set of dependency rules, and checks whether the rules make the test crash consistent according to Definition 5. This procedure uses symbolic evaluation of the storage system implementation \mathcal{O} to generate the logical encoding described in Section 3.3, and solves the resulting formulas using an off-the-shelf SMT solver [20].

4.3 Synthesizing Dependency Rules with Happens-Before Graphs

The core of the DEPSYNTH algorithm is the RULESFORTEST procedure in Figure 4, which takes as input a litmus test T , a storage system implementation \mathcal{O} , and a crash-consistency predicate *Consistent*, and synthesizes a set of dependency rules that makes T crash consistent. RULESFORTEST frames the rule synthesis problem as a search over *happens-before graphs* [14] on the writes performed by the test. An edge (w_1, w_2) between two writes in a happens-before graph says that write w_1 must persist to disk before write w_2 . Happens-before graphs and

```

22 function RULESFORTTEST( $T = \langle P_{initial}, P_{main} \rangle, \mathcal{O}, Consistent$ )
23    $W \leftarrow \{w \mid w \in Evaluate_{\mathcal{O}}(P_{main})\}$ 
24   return PHASE1( $\mathcal{T}, [], W, \mathcal{O}, Consistent$ )

25 function PHASE1( $T, order, W, \mathcal{O}, Consistent$ ) ▷ Search for total orders over writes
26   if  $W = \emptyset$  then
27      $G \leftarrow \{(order[i], order[j]) \mid 0 \leq i < j < |order|\}$ 
28     return PHASE2( $\mathcal{T}, G, \mathcal{O}, Consistent$ ) ▷  $G$  is a total order; minimize it in Phase 2
29   for  $w \in W$  do
30      $order' \leftarrow order + [w]$ 
31      $W' \leftarrow W \setminus \{w\}$ 
32      $G \leftarrow \{(order[i], order[j]) \mid 0 \leq i < j < |order|\} \cup$ 
33        $\{(w_1, w_2) \mid w_1 \in order \wedge w_2 \in W\} \cup$ 
34        $\{(w_1, w_2) \mid w_1, w_2 \in W\}$ 
35     if  $\neg CRASHCONSISTENT(T, RULESFORGRAPH(G), \mathcal{O}, Consistent)$  then
36       continue
37      $R \leftarrow PHASE1(T, order', W', \mathcal{O}, Consistent)$ 
38     if  $R \neq \perp$  then
39       return  $R$ 
40   return  $\perp$ 

39 function PHASE2( $T, G, \mathcal{O}, Consistent$ ) ▷ Minimize graph  $G$  by removing individual edges
40    $R \leftarrow RULESFORGRAPH(G)$ 
41   if  $\neg CRASHCONSISTENT(T, R, \mathcal{O}, Consistent)$  then
42     return  $\perp$ 
43   for  $(w_1, w_2) \in G$  do ▷ Try removing each edge from  $G$ 
44      $G' \leftarrow G \setminus \{(w_1, w_2)\}$ 
45      $R' \leftarrow PHASE2(T, G', \mathcal{O}, Consistent)$ 
46     if  $R' \neq \perp$  then
47       return  $R'$ 
48   if  $ACYCLIC(R)$  then
49     return  $R$  ▷  $G$  makes  $T$  crash consistent and no subgraph of  $G$  suffices
50   else
51     return  $\perp$ 

52 function RULESFORGRAPH( $G$ ) ▷ Generalize a happens-before graph into dependency rules
53    $R \leftarrow \{\}$ 
54   for  $(w_1, w_2) \in G$  do
55      $\langle n_1, t_1 \rangle \leftarrow LABEL(w_1)$  ▷ Get label  $l_1 = \langle n_1, t_1 \rangle$  for write  $w_1 = write(a_1, s_1, l_1)$ 
56      $\langle n_2, t_2 \rangle \leftarrow LABEL(w_2)$ 
57     if  $t_1 < t_2$  then
58        $R \leftarrow R \cup \{n_2 \rightsquigarrow_{>} n_1\}$  ▷ Invert order, as a rule  $n_a \rightsquigarrow_p n_b$  says  $n_a$  happens after  $n_b$ 
59     else if  $t_1 = t_2$  then
60        $R \leftarrow R \cup \{n_2 \rightsquigarrow_{=} n_1\}$ 
61     else
62        $R \leftarrow R \cup \{n_2 \rightsquigarrow_{<} n_1\}$ 
63   return  $R$ 

```

■ **Figure 4** The algorithm for generating sufficient dependency rules for a litmus test T searches the space of happens-before graphs over the writes performed by T . The first phase searches for total orders over the writes that are sufficient for crash consistency. Once such a total order is found, the second phase removes edges from it until the happens-before graph is minimal.

dependency rules have a natural correspondence: if a happens-before graph includes an edge (w_1, w_2) , a dependency rule $n_2 \rightsquigarrow_p n_1$ that matches the writes' labels is sufficient to enforce the required ordering. RULESFORTTEST searches for a minimal, acyclic happens-before graph that is sufficient to ensure crash consistency for T , and then syntactically generalizes that happens-before graph into a set of dependency rules.

RULESFORTTEST searches for a happens-before graph by first finding a *total order* on the writes that makes T crash consistent (PHASE1), and then searching for a minimal *partial order* within this total order that is both sufficient for crash consistency and yields an acyclic set of dependency rules (PHASE2). The algorithm is exhaustive: it tries all total orders and all minimal partial orders within a total order, until it finds a solution or fails because a solution does not exist.

RULESFORTTEST builds on the observation that crash consistency (Definition 5) is monotonic with respect to the subset relation on dependency rules – if a set of dependency rules R is not sufficient for crash consistency, then no subset of R is sufficient either:

► **Theorem 7** (Monotonicity of crash consistency). *Let T be a litmus test and R a set of dependency rules for a storage system \mathcal{O} . If R does not make T crash consistent (according to Definition 5), then no subset $R' \subset R$ can make T crash consistent.*

Proof sketch. If R does not make T crash consistent, there exists a valid crash schedule \mathbf{s} (Definition 4) that does not satisfy the crash consistency predicate *Consistent*. By Definition 4, each rule in R only adds additional constraints on the possible valid crash schedules. Removing a rule from R therefore only allows more valid crash schedules, and so if \mathbf{s} was a valid crash schedule for R , it is also a valid crash schedule for any subset of R . ◀

RULESFORTTEST applies this property by checking crash consistency for a happens-before graph G before exploring any subgraphs of G ; if G is not sufficient, then neither is any subgraph of G , and so that branch of the search can be skipped.

4.3.1 Total order search

PHASE1 (line 25) explores all possible total orders over the writes in T that are sufficient for crash consistency. At each recursive call, the list *order* represents a total order over some of T 's writes, and the set W contains all writes not yet added to that order. PHASE1 tries to add each write in W to the end of the total order. Each time, it checks whether the new total order leads to a crash consistency violation (line 33) and if so, prunes this branch of the search. For PHASE1 to be complete, this check must behave angelically for the writes in W that have not yet been added to the order – if there is *any possible* set of dependency rules for the remaining writes that would succeed, the check must succeed. We make the check angelic by including every possible dependency rule for the remaining writes (line 32). If the test cannot be made crash consistent even with every possible rule included, then by Theorem 7 no subset of those rules (i.e., formed by completing the rest of the total order) can succeed either, so the prefix is safe to prune. PHASE1 continues until every write has been added to the total order and then moves to PHASE2 to further reduce the happens-before graph.

4.3.2 Partial order search

Starting from a happens-before graph G that reflects a total order over all writes in T , PHASE2 (line 39) removes edges from the graph until it is minimal, i.e., removing any further edges would violate crash consistency. PHASE2 removes one edge at a time from the graph

G (line 44), checks if the graph remains sufficient for crash consistency (line 41), and if so, recurses to remove more edges. By greedily removing one edge at a time, PHASE2 is guaranteed to find a minimal result, and because PHASE2 considers removing every possible edge from G (except those that cannot lead by solutions by Theorem 7), it is complete – if an acyclic solution exists, PHASE2 will reach it.

4.3.3 Generating rules from happens-before graphs

The RULESFORTTEST search operates on happens-before graphs, but its goal is to synthesize dependency rules (Definition 1). The RULESFORGRAPH procedure (line 52) bridges this gap by taking as input a happens-before graph G and returning a set of dependency rules R that are sufficient to enforce the ordering requirements that G dictates. RULESFORGRAPH uses a simple syntactic approach to generate a rule for each edge in G : if $(w_1, w_2) \in G$, where w_1 and w_2 have labels $l_1 = \langle n_1, t_1 \rangle$ and $l_2 = \langle n_2, t_2 \rangle$, respectively, then it generates a rule of the form $n_2 \rightsquigarrow n_1$ (reversing the order because G is a happens-before graph but dependency rules are happens-after edges). To choose an epoch predicate for the generated rule, we compare the two epochs t_1 and t_2 and select the predicate that would make the rule match the labels l_1 and l_2 .

This approach can lead to rules that are too general, as some rules it generates may only need to apply to certain individual epochs but will instead apply to all epochs that match the predicate. Overly general rules risk sacrificing performance by preventing reordering or caching optimizations that would be safe. However, this same generality also allows RULESFORTTESTS to avoid overfitting to the input litmus tests. In Section 6.1 we show that generated rules generalize well in practice (i.e., are not overfit), and that they filter out few additional schedules compared to expert-written rules.

4.3.4 Properties of RulesForTest

The RULESFORTTEST algorithm is *sound*: all paths that return a solution are guarded by checks of crash consistency and of acyclicity, and so satisfy the requirements of Section 4.1. RULESFORTTEST is also *complete*: each of PHASE1 and PHASE2 are complete, as discussed above, and so together form a complete search over the space of total orders. Every possible acyclic solution must be a subgraph of some total order, since the transitive closure of edges in any happens-before graph is a (strict) partial order, and so exploring all total orders suffices to reach any possible acyclic solution. Finally, RULESFORTTEST is *minimal*, in the sense that removing any rule from a returned set R would violate crash consistency. PHASE2 continues removing edges from a candidate graph G until Theorem 7 says it cannot be made smaller, and is therefore guaranteed to find a minimal happens-before graph. Every rule in R is justified by (at least) one edge in that graph, and since dependency rules cannot overlap (in Definition 1, the possible epoch predicates are disjoint), removing any rule would incorrectly allow reordering of its corresponding edge(s).

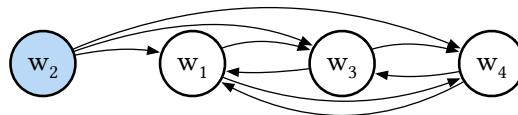
► **Example 8.** Consider running RULESFORTTEST for the simple log-structured key-value store and `SingleEntry_TwoAppend` litmus test from Section 2. From Example 6 we know that this test produces a set W of four writes:

```

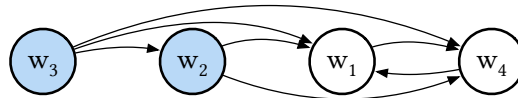
w1 = write(2, to_block((1, 81)), ⟨log, 1⟩),
w2 = write(0, to_block((1, 3)), ⟨superblock, 1⟩),
w3 = write(3, to_block((2, 37)), ⟨log, 2⟩),
w4 = write(0, to_block((1, 4)), ⟨superblock, 2⟩)

```

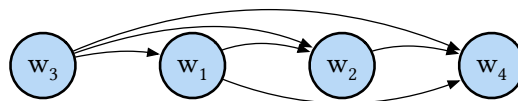
PHASE1 first chooses the first write to add to the total order. Suppose it chooses w_2 . This choice results in the following graph G at line 32 (shaded nodes are in *order*; white nodes are in W):



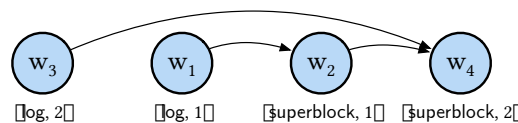
The check at line 33 finds that this graph is not crash consistent: it allows a crash schedule where w_2 is on disk but no other writes are, which violates the crash-consistency predicate as w_2 is a superblock write pointing to a log block that is not on disk. PHASE1 therefore continues (line 34), which prunes any total order that starts with $[w_2]$ from the search, and chooses a next write to consider, say w_3 . The total order starting with $[w_3]$ does pass the crash consistency check, so PHASE1 recurses with $order = [w_3]$ and $W = \{w_1, w_2, w_4\}$. In this recursive call, suppose we again first choose w_2 to add to the total order. This choice results in the following graph G :



Again, line 33 finds that this graph is not crash consistent, for the same reason as before (superblock write w_2 can be on disk when log write w_1 is not), and so the search continues, pruning any total order that starts with $[w_3, w_2]$. Suppose it next chooses w_1 to add to the total order. This choice succeeds, making the recursive call with $order = [w_3, w_1]$ and $W = \{w_2, w_4\}$. From here, any choice PHASE1 makes will succeed. Supposing it chooses w_2 first, PHASE1 eventually reaches line 28 and continues to PHASE2 with the following initial graph G :



PHASE2 proceeds by trying to remove one edge at a time from G . Suppose it first chooses to remove edge (w_3, w_1) , and so recurses at line 45 on the graph $G' = G \setminus \{(w_3, w_1)\}$. This graph still ensures crash consistency at line 41, as writing w_1 before w_3 does not affect consistency. The recursion can continue twice more by choosing and successfully removing edges (w_3, w_2) and then (w_1, w_4) as well, eventually reaching line 43 with the following graph G (now with write labels shown):



From here, the loop in PHASE2 now tries to remove each of the three remaining edges, but each attempted G' violates crash consistency and so returns \perp from the next recursive call. PHASE2 therefore exits the loop with the above graph G , which we now know is minimal as no further edges can be removed. Applying RULESFORGRAPH to G yields the two rules from Section 2:

$$\begin{array}{ll} \text{superblock} \rightsquigarrow_{=} \text{log} & \text{from edges } (w_3, w_4) \text{ and } (w_1, w_2) \\ \text{superblock} \rightsquigarrow_{>} \text{superblock} & \text{from edge } (w_2, w_4). \end{array}$$

4.4 Resolving Cycles in Dependency Rules

The top-level DEPSYNTH algorithm generates rules for each litmus test independently. Even though the rules generated for each test are guaranteed to be acyclic, it is possible for the *union* of those rules to contain a cycle, and so violate the requirements of Section 4.1. In practice, we have not seen this happen for the storage systems we evaluate in Section 6, and so the version of DEPSYNTH presented in Figure 3 fails if the synthesized rules contain a cycle.

To handle cyclic rules, RULESFORTEST can be extended to support synthesizing rules for multiple litmus tests at once. This extension adds the writes from *all* the tests into the set of writes W , searches for a total order over that entire set in PHASE1, and then searches for a minimal happens-before graph over the entire set in PHASE2. Edges between writes from different tests cannot influence the crash consistency of individual tests (in Definition 4 they will just lead to spurious additional implications), and they will eventually be removed by PHASE2, creating a forest of disjoint happens-before graphs. PHASE2 is therefore guaranteed to return an acyclic set of dependency rules for all the tests it was provided.

In the limit, DEPSYNTH could just invoke RULESFORTEST with its entire input set \mathcal{T} , but this would be prohibitively expensive for any non-trivial set of tests. Instead, our implementation resolves cycles in DEPSYNTH by identifying which individual litmus tests caused the cycle (i.e., which tests the rules in the cycle were generated from), and passes only that subset of tests to the extended RULESFORTEST.

5 Implementation

We implement both the DepSynth algorithm and the storage systems we study in Section 6 in Rosette [28], an extension of Racket [9] with support for verification and synthesis. Using Rosette as our host language gives us symbolic evaluation of the storage system implementation for free, and simplifies implementing the CRASHCONSISTENT query in Figure 3. The choice of Rosette and Racket is not fundamental; recent work has shown how to extend the symbolic evaluation approach to languages such as Python [27] or C [19] in which storage systems are more commonly implemented.

5.1 Ordering

The DEPSYNTH algorithm in Figure 3 is sensitive to the order in which NEXTTEST chooses tests to generate dependency rules for. Our implementation chooses tests in increasing order of size, minimizing the number of happens-before graphs for RULESFORTEST to explore. Similarly, RULESFORTEST is sensitive to the order it considers writes (PHASE1) and edges (PHASE2). In both cases, we exploit the following observation: while an execution that persists writes in program order is not *required* to be crash consistent (e.g., because storage

systems might selectively buffer or coalesce writes), it is often so in practice. `RULESFORTTEST` therefore prefers to choose writes in `PHASE1` in program order, and prefers to remove edges in `PHASE2` that contradict program order.

5.2 Reducing solver queries

Both `PHASE1` and `PHASE2` in Figure 4 have symmetry in their search space: for a fixed pair of writes w_1 and w_2 , there are many different branches of `PHASE1` that try to order w_2 after w_1 , and many different branches of `PHASE2` that try to remove the edge (w_1, w_2) from a happens-before graph. If we can determine ahead of time that such a choice for those writes is always doomed to fail, we can avoid considering these choices at all and so save the cost of an SMT solver query by `CRASHCONSISTENT`. Our implementation of `RULESFORTTEST` uses an SMT solver to pre-compute a set of *necessary* ordering edges – edges which *must* be in the happens-before graph – and uses that set to short-circuit `CRASHCONSISTENT`.

6 Evaluation

This section answers three questions to demonstrate the effectiveness of DepSynth:

1. Can storage system developers use DepSynth to synthesize dependency rules for a realistic storage system rather than implementing their own crash-consistency approach by hand? (§6.1)
2. Can DepSynth help storage system developers avoid crash-consistency bugs? (§6.2)
3. Does DepSynth’s approach support a variety of storage system designs? (§6.3)

6.1 ShardStore Case Study

To show that developers can use DepSynth to build realistic storage systems, we implemented a key-value store that follows the design of ShardStore [4], the exabyte-scale production storage node for the Amazon S3 object storage service.

6.1.1 Implementation

The first step in using DepSynth is to implement the storage system itself. ShardStore’s on-disk representation is a log-structured merge tree (LSM tree) [21], but with values stored outside the tree in a collection of extents. Our ShardStore-like storage system implementation consists of 1,200 lines of Racket code, including five operations: the usual `put`, `get`, and `delete` operations on single keys, as well as a garbage collection `clean` operation that evacuates all live objects in one extent to another extent, and a `flush` operation that persists the LSM tree memtable to disk. Our implementation does not handle boundary conditions such as running out of disk space or objects too large to fit in one extent, but is otherwise faithful to the published ShardStore design. As a crash consistency predicate, we wrote a checker that validates all expected objects are accessible by `get` after a crash, and that the on-disk LSM tree contains only valid pointers to objects in extents.

6.1.2 Synthesis

With a storage system implementation in hand, a developer can use DepSynth to synthesize dependency rules that make the system crash consistent. DepSynth takes as input a set of litmus tests – we randomly generated 16,250 litmus tests for the ShardStore-like system, ranging in length from 1 to 16 operations. Executing these tests against the system led to

■ **Table 1** Valid schedules allowed by the production ShardStore service versus the dependency rules we synthesized for our ShardStore-like reimplementation. A schedule allowed only by one implementation means either that implementation is not crash consistent (it allows a schedule it should forbid) or it admits more reordering opportunities (it allows a schedule it should allow). “Fixed” results are after fixing two issues in ShardStore (one consistency, one performance) that we identified by manually inspecting the “original” schedules.

Test	Test Length	Writes	Allowed by both		Allowed only by DepSynth		Allowed only by ShardStore	
			Original	Fixed	Original	Fixed	Original	Fixed
T_1	1	2	3	3	0	0	0	0
T_2	2	6	7	14	7	0	3	3
T_3	5	1	2	2	0	0	0	0
T_4	5	7	8	15	7	0	3	3
T_5	4	7	11	29	9	0	9	0
T_6	5	5	6	12	2	0	4	0
T_7	7	5	5	11	2	0	5	1
T_8	10	5	6	12	2	0	4	0
T_9	16	6	8	22	2	0	12	0
T_{10}	13	9	21	41	20	0	9	9

an average of 7.2 and a maximum of 20 disk writes per test. Given these inputs, DepSynth synthesized a set of 20 dependency rules for ShardStore in 49 minutes. To find a correct solution for all 16,250 litmus tests, the DepSynth algorithm invoked the RULESFORTTEST procedure (line 8 in Figure 3) only 10 times, showing that DepSynth’s incremental approach is effective at reducing the search space.

6.1.3 Comparison to an existing implementation

ShardStore is an existing production system and already supports crash consistency. Its implementation does not use a dependency-rule language like in DepSynth. Instead, it implements a soft-updates approach [11] by constructing dependency graphs (i.e, happens-before graphs) at run time and sequencing writes to disk based on those graphs, similar to patchgroups in Featherstitch [10]. We therefore compare our synthesized rules against ShardStore’s dependency graphs to see how well DepSynth may replace an expert-written crash consistency implementation.

For each of the 10 tests that DepSynth used while synthesizing dependency rules for ShardStore, we used an SMT solver to compute the set of valid crash schedules (Definition 4) according to those synthesized dependency rules. We then executed the same test using the production ShardStore implementation, collected the run-time dependency graph it generated, and used an SMT solver to compute the set of valid crash schedules according to that graph. Given these two sets of crash schedules, we computed the set intersection and difference to classify them into three groups: schedules allowed by both implementations (i.e., both implementations agree), and schedules allowed only by one or the other implementation (i.e., the two implementations disagree).

Table 1 shows the results of this classification across the 10 litmus tests. Overall, the two implementations agree on the validity of an average of 87% of crash schedules. The remaining crash schedules are in two categories:

1. Schedules allowed only by DepSynth mean either DepSynth’s rules allow some schedules that are not crash consistent (a correctness issue in the synthesized rules) or ShardStore precludes some schedules that are crash consistent (a performance issue in ShardStore).

We found that every schedule allowed by DepSynth is crash consistent, and that ShardStore inserts unnecessary edges in its dependency graphs, ruling out some reorderings that would be safe. These edges are not necessary to guarantee crash consistency of the overall storage system, and so DepSynth is correct to allow them. However, ShardStore engineers intentionally include these edges as they make the representation invariant for an on-disk data structure simpler, even though a more complex invariant that did not require these edges would still be sufficient for consistency. In other words, ShardStore engineers favored a stronger, simpler invariant in these cases, where DepSynth is able to identify opportunities for performance improvements.

2. Schedules allowed only by ShardStore mean either DepSynth’s rules preclude some schedules that are crash consistent (meaning DepSynth’s output is not optimal) or ShardStore allows some schedules that are not crash consistent (a correctness issue in ShardStore). 67% of these schedules are incorrectly allowed by ShardStore due to a rare crash-consistency issue that was independently discovered concurrently with this work. We have confirmed with ShardStore engineers that the issue was an unlikely edge case that could not lead to data loss, but could lead to “ghost” objects – resurrected pointers to deleted objects, where the object data has been (correctly) deleted, but the pointer still exists – which result in an inconsistent state. After fixing this issue in ShardStore, we manually inspected the remaining schedules it allowed and confirmed they are all cases where DepSynth’s rules generate extraneous edges (i.e., the synthesized rules are not optimal), and the crash-consistency predicate we wrote for our ShardStore reimplementation agrees that all the resulting states are consistent.

After fixing the two ShardStore issues discussed above, the synthesized dependency rules agree with ShardStore on the validity of an average of 99% of crash schedules. The few remaining schedules are ones that DepSynth’s synthesized dependency rules conservatively forbid due to the coarse granularity of the dependency rule language. Overall, this study shows that DepSynth achieves similar results to an expert-written crash consistency implementation, and can help identify correctness and performance issues in existing storage systems.

6.1.4 Generalization

One risk for example-guided synthesis techniques like DepSynth is that they can overfit to the examples (litmus tests) and not actually ensure crash consistency on unseen test cases. DepSynth’s design reduces this risk by using a simple dependency rule language (Definition 1) that cannot identify individual write operations. To test generalization, we randomly generated an additional 136,000 litmus tests for our ShardStore-like system. We also allowed these tests to be significantly longer than those used during synthesis – up to a maximum of 40 writes rather than the 20 in the input set of litmus tests. For each new test, we used the synthesized dependency rules to compute all valid crash schedules for the test, and found that every crash schedule resulted in a consistent disk state according to our crash consistency predicate. In other words, by limiting the expressivity of our dependency rule language, the rules we synthesize can generalize well beyond the tests they were generated from.

6.2 Crash-Consistency Bugs

To understand how effective DepSynth can be in preventing crash-consistency bugs, we surveyed all bugs reported by two recent papers [4, 18] in three production storage systems for which a known fix is available. We manually analyze each bug and determine whether DepSynth could discover and prevent them.

■ **Table 2** Sample crash-consistency bugs in three storage systems reported by two recent papers [4, 18]. Each bug includes its identifier (bug number for ShardStore, kernel Git commit for btrfs and f2fs). Most of these bugs could have been prevented by using DepSynth to automatically identify missing ordering requirements, but some crash-consistency issues are either not ordering related or are unlikely to be detected by DepSynth’s litmus-test-driven approach.

Storage system	Crash-consistency bug	Preventable by DepSynth?
ShardStore	Inconsistency in extent allocation (#6)	Yes
ShardStore	Mismatch between soft and hard write pointers (#7)	Yes
ShardStore	Index entries persisted before target data (#8)	Yes
ShardStore	Crash consistency predicate too strong (#9)	No – specification bug
ShardStore	Data loss after UUID collision (#10)	No – unlikely to detect
btrfs	Extents deallocated too early in recovery (bf50411)	Yes
btrfs	Inode rename commits out of order (d4682ba)	Yes
f2fs	<code>fsync</code> failed after directory rename (ade990f)	Yes
f2fs	Wrong file size when zeroing file beyond EOF (17cd07a)	No – not reordering

Table 2 shows the results of our survey. In six cases, DepSynth could have prevented the bug by synthesizing a dependency rule to preclude a problematic reordering optimization. Each of these bugs had small triggering test cases, suggesting they would be reachable by a litmus-test-based approach like ours. In the other three cases, our analysis shows that DepSynth would not prevent the bug. One bug in ShardStore was a specification bug in which the crash consistency predicate was too strong. DepSynth assumes that the crash consistency predicate is correct, and will miss specification bugs. Another bug in ShardStore involved a collision between two randomly generated UUIDs. While such a bug would be possible to find in principle using litmus tests, it would be very unlikely, and without a test that triggers the issue DepSynth cannot preclude it. One bug in f2fs involved an incorrect file size being computed when zero-filling a file beyond its existing endpoint. This bug was a logic issue rather than a reordering one (i.e., occurring even without a crash), and so no dependency rule would suffice to prevent it. Overall, our analysis indicates that DepSynth can prevent a range of ordering-related crash-consistency bugs, but other bugs would require a different approach.

6.3 Other Storage Systems

Beyond ShardStore, we expect DepSynth to effectively generate rules for any storage systems whose crash consistency properties can be ensured by correctly ordering writes. As Frost et al. describe in [10], write-before relationships underlie *every* crash consistency mechanism, including journaling, synchronous writes, copy-on write data structures, and soft updates. Though storage systems vary greatly in their mechanisms for storing and retrieving data, each must enable crash consistency by enforcing write-before relationships. Since DepSynth is a tool for automatically developing write-before relationships, this leads us to believe that DepSynth can be a useful tool for automating crash consistency in all such systems.

To demonstrate this point, we have also used DepSynth to implement a log-structured file system [25]. The file system supports five standard POSIX operations: `open`, `creat`, `write`, `close`, and `mkdir`. While our implementation is simple (300 lines of Racket code) compared to production file systems, it has metadata structures for files and directories, and so has its own subtle crash consistency requirements. For example, updates to data and inode blocks must reach the disk before the pointer to the tail of the log is updated. To synthesize dependency rules for this file system, we randomly generated 235 litmus tests with at most 6 operations. DepSynth synthesized a set of 18 dependency rules in 12 minutes

to make the file system crash consistent, and during the search, invokes `RULESFORTTEST` for only 13 tests. This result shows that DepSynth can automate crash consistency for storage systems other than key-value stores.

7 Related Work

7.1 Verified storage systems

Inspired by successes in other systems verification problems [16, 13], recent work has brought the power of automated and interactive verification to bear on storage systems as well. One of the main challenges in verifying storage systems is crash consistency, as it combines concurrency-like nondeterminism with persistent state. Yggdrasil [27] is a verified file system whose correctness theorem is a *crash refinement* – a simulation between a crash-free specification and the nondeterministic, crashing implementation. This formalization allows clients of Yggdrasil to program against a strong specification free from crashes, similar to our angelic crash consistency model. FSCQ [8] is a verified crash-safe file system with specifications stated in *crash Hoare logic*, which explicitly states the recovery behavior of the system after a crash. DFSCQ [7] extends FSCQ and its verification with support for crash-consistency optimizations such as log-bypass writes and the metadata-only `fdatasync` system call. The DepSynth programming model separates crash consistency of these optimizations from the storage system itself, and so can simplify their implementation.

Another approach to verified storage systems is at the language level. Cogent [3] is a language for building storage systems with a strong type system that precludes some common systems bugs. A language-level approach like Cogent is complementary to DepSynth: Cogent provides a high-level language for implementing storage systems, while DepSynth provides a synthesizer for making those implementations crash consistent.

7.2 Crash-consistency bug-finding tools

Ferrite [5] is a framework for specifying *crash-consistency models*, which formally define the behavior of a storage system across crashes, and for automatically finding violations of such models in a storage system implementation. One way to specify these models is with litmus tests that demonstrate unintuitive behaviors; DepSynth builds on this approach by automatically synthesizing rules from such litmus tests. DepSynth also takes inspiration from Ferrite’s synthesis tool for inserting `fsync` calls into litmus tests to make them crash consistent, but instead focuses on making the *storage system itself* crash consistent rather than the user code running on top of it. CrashMonkey [18] is a tool for finding crash-consistency bugs in Linux file systems. CrashMonkey exhaustively enumerates all litmus tests with a given set of system calls, runs them against the target file system, and then tests each possible crash state for consistency. Chipmunk [15] extends the CrashMonkey approach to persistent-memory file systems by exploring finer-grained crash states to account for the byte-addressable nature of non-volatile memory. Connecting CrashMonkey-like litmus test generation with DepSynth could provide developers with a comprehensive set of litmus tests for their system for free, lowering the burden of applying DepSynth. To give stronger coverage guarantees that do not depend on enumerating litmus tests, FiSC [33] and eXplode [34] use model checking to find bugs in storage systems.

One advantage of bug-finding tools is that they are significantly easier to apply to production systems than heavyweight verification tools. Bornholt et al. [4] describe the use of lightweight formal methods to validate the crash consistency (and other properties) of

ShardStore, the Amazon S3 storage node that we study in Section 6.1. Their approach applies property-based testing to automatically find and minimize litmus tests that demonstrate crash-consistency issues. DepSynth takes this idea one step further by automatically *fixing* such issues once they are found.

7.3 Program synthesis for systems code

Transit [31] is a tool for automatically inferring distributed protocols such as those used for cache coherence. It guides the search using *concolic* snippets [26] – effectively litmus tests that can be partially symbolic – and finds a protocol that satisfies those snippets for *any* ordering of messages. MemSynth [6] is a program synthesis tool for automatically constructing specifications of memory consistency models. MemSynth takes similar inputs to DepSynth – a set of litmus tests and a target language – and its synthesizer generates and checks happens-before graphs for those tests. Adopting MemSynth’s aggressive inference of partial interpretations [29] to shrink the search space of happens-before graphs would be promising future work.

8 Conclusion

DepSynth offers a new programming model for building crash-consistent storage systems. By offering a high-level angelic programming model for crash consistency, and automatically synthesizing low-level dependency rules to realize that model, DepSynth lowers the burden of building reliable storage systems. We believe that this work presents a promising direction for building systems software with the aid of automatic programming tools to resolve challenging nondeterminism and persistence problems.


References

- 1 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, pages 258–272, Edinburgh, United Kingdom, July 2010.
- 2 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Saarbrücken, Germany, March–April 2011.
- 3 Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O’Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188, Atlanta, GA, April 2016.
- 4 James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual conference, October 2021.
- 5 James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98, Atlanta, GA, April 2016.

- 6 James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 467–481, Barcelona, Spain, June 2017.
- 7 Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 270–286, Shanghai, China, October 2017.
- 8 Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, October 2015.
- 9 Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, March 2018.
- 10 Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 307–320, Stevenson, WA, October 2007.
- 11 Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–60, Monterey, CA, November 1994.
- 12 Valerie Henson. The many faces of fsck, September 2007. URL: <https://lwn.net/Articles/248180/>.
- 13 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Michael Norrish, Rafal Kolanski, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, October 2009.
- 14 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.
- 15 Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. Chipmunk: Investigating crash-consistency in persistent-memory file systems. In *Proceedings of the 18th ACM EuroSys Conference*, Rome, Italy, May 2023.
- 16 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.
- 17 Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, February 2013.
- 18 Jayashree Mohan, Ashlie Martinez, Soujanya Ponnappalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–50, Carlsbad, CA, October 2018.
- 19 Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 225–242, Huntsville, Ontario, Canada, October 2019.
- 20 Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 9:53–58, 2015.
- 21 Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, June 1996.

- 22 Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnathan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 433–448, Broomfield, CO, October 2014.
- 23 Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 105–120, Anaheim, CA, April 2005.
- 24 Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage*, 9(3):9:1–32, August 2013.
- 25 M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, October 1991.
- 26 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 13th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lisbon, Portugal, September 2005.
- 27 Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, November 2016.
- 28 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 530–541, Edinburgh, United Kingdom, June 2014.
- 29 Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Braga, Portugal, March–April 2007.
- 30 Stephen C. Tweedie. Journaling the Linux ext2fs filesystem. In *Proceedings of the 4th Annual LinuxExpo*, Durham, NC, May 1998.
- 31 Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 287–296, Seattle, WA, June 2013.
- 32 John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 2017.
- 33 Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, San Francisco, CA, December 2004.
- 34 Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, November 2006.
- 35 Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 449–464, Broomfield, CO, October 2014.


Synthesizing Conjunctive Queries for Code Search

Chengpeng Wang ✉ 

The Hong Kong University of Science and Technology, China

Peisen Yao ✉ 

Zhejiang University, Hangzhou, China

Wensheng Tang ✉ 

The Hong Kong University of Science and Technology, China

Gang Fan ✉ 

Ant Group, Shenzhen, China

Charles Zhang ✉ 

The Hong Kong University of Science and Technology, China

Abstract

This paper presents SQUID, a new conjunctive query synthesis algorithm for searching code with target patterns. Given positive and negative examples along with a natural language description, SQUID analyzes the relations derived from the examples by a Datalog-based program analyzer and synthesizes a conjunctive query expressing the search intent. The synthesized query can be further used to search for desired grammatical constructs in the editor. To achieve high efficiency, we prune the huge search space by removing unnecessary relations and enumerating query candidates via refinement. We also introduce two quantitative metrics for query prioritization to select the queries from multiple candidates, yielding desired queries for code search. We have evaluated SQUID on over thirty code search tasks. It is shown that SQUID successfully synthesizes the conjunctive queries for all the tasks, taking only 2.56 seconds on average.

2012 ACM Subject Classification Software and its engineering → Automatic programming; Human-centered computing → User interface programming

Keywords and phrases Query Synthesis, Multi-modal Program Synthesis, Code Search

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.36

Related Version *Full Version*: <https://arxiv.org/abs/2305.04316>

Funding The authors are supported by the RGC16206517, ITS/440/18FP and PRP/004/21FX grants from the Hong Kong Research Grant Council and the Innovation and Technology Commission, Ant Group, and the donations from Microsoft and Huawei.

Acknowledgements We thank the anonymous reviewers, Xiao Xiao, and Xiaoheng Xie for their helpful comments. Peisen Yao is the corresponding author.

1 Introduction

Developers often need to search their code for target patterns in various scenarios, such as API understanding [29], code refactoring [56], and program repair [47]. According to recent studies [34, 30], existing efforts have to compromise between ease of use and capability. Most mainstream IDEs [23] only support string match or structural search of restrictive grammatical constructs although complex user interactions are not required. Besides, static program analyzers, such as Datalog-based program analyzers [44, 35, 3], provide deep program facts for users to explore advanced patterns, while users have to customize the analyzers to meet their needs [12]. For example, if users want to explore code patterns with the Datalog-based program analyzer CODEQL [3], they have to learn the query language to access the derived relational representation. However, there always exists a non-trivial gap



© Chengpeng Wang, Peisen Yao, Wensheng Tang, Gang Fan, and Charles Zhang;
licensed under Creative Commons License CC-BY 4.0

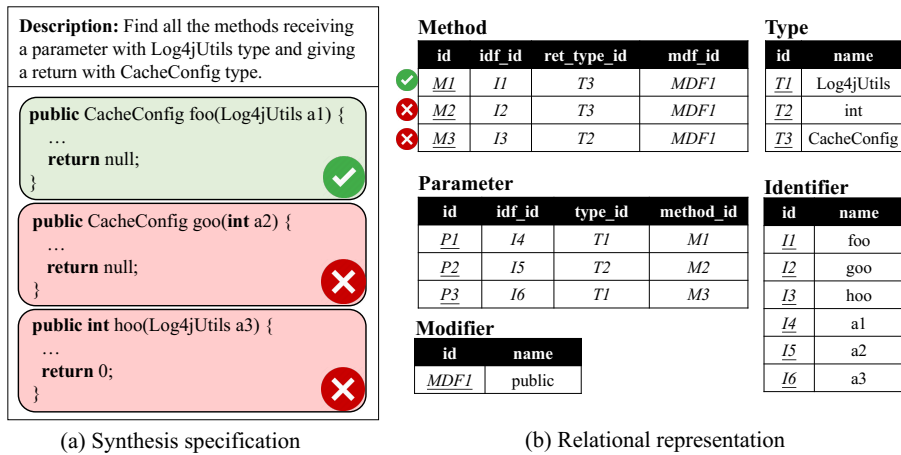
37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 36; pp. 36:1–36:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Target(id, idf1, retTypeId, mdf) :-

Method(id, idf1, retTypeId, mdf), **Type**(retTypeId, name1), **equal**(name1, "CacheConfig"),

Parameter(pld, idf2, pTypeId, id), **Type**(pTypeId, name2), **equal**(name2, "Log4jUtils")

(c) Conjunctive query

■ **Figure 1** A motivating example¹.

between a user's search intent and a customized query. A large number of complex relations make query writing involve strenuous efforts, especially in formalizing search intents and debugging queries, which hinders the usability of CODEQL for code search.

Our Goal. We aim to propose a query synthesizer to unleash the power of a Datalog-based program analyzer for code search. To show the search intent, a user can specify a synthesis specification consisting of positive examples, negative examples, and a natural language description. Specifically, positive and negative examples indicate desired and non-desired grammatical constructs, respectively, while the natural language description shows the search intent by a sentence. Our synthesizer is expected to generate a query separating positive examples from negative ones, which can support code search in the editor. In this work, we focus on conjunctive queries, which have been recognized as queries of an important form to support search tasks [17].

Consider a usage scenario: Find all the methods receiving a parameter with `Log4jUtils` type and giving a return with `CacheConfig` type. The user can provide the synthesis specification shown in Figure 1(a). With the relational representation in Figure 1(b) derived from the examples, our synthesizer would synthesize the conjunctive query in Figure 1(c) to express the search intent. In particular, our synthesis specification is easy to provide. The users can often copy desired grammatical constructs from an editor as positive examples [34] and then mutate them to form negative ones. Meanwhile, they can express their need to search code with a brief sentence as the description. Thus, an effective and efficient synthesizer enables the users to express the search intent from a high-level perspective, serving as a user-friendly interface for code search.

¹ We show five relations as examples, while a Datalog-based analyzer can derive over a hundred relations.

Challenges. Nevertheless, it is far from trivial to synthesize a conjunctive query for code search. First, a Datalog-based program analyzer can generate many relations with multiple attributes as the relational representation. For example, CODEQL exposes over a hundred relations to users for query writing [45]. The various choices of selecting relations and enforcing conditions on attributes induce a dramatically huge search space in the synthesis, which can involve both the comparisons between attributes and string constraints, posing a significant challenge to achieving high efficiency. Second, there often exist multiple query candidates that separate positive examples from negative ones, while several candidates can suffer from the over-fitting problem, failing to express the search intent with no bias [53]. An ineffective query candidate selection would mislead the synthesizer into returning wrong queries and further cause the failure of code search.

Existing Effort. There are three major lines of existing effort. The first line of the studies utilizes input-output examples to synthesize queries in various forms, such as analytic SQL queries [58, 13] and relational queries [42, 46]. Although the queries often have expressive syntax, the synthesizers only take a few relations as input, not facing hundreds of relations as ours. The second line of approaches is the component-based synthesis technique [14, 38], which leverages type signatures to enumerate well-typed programs. However, a significant number of comparable attribute pairs still induce an explosively huge search space even if we adopt the techniques by guiding the search with the schema. The third line of the studies derives program sketches from natural language descriptions via semantic parsing [28] and prioritizes feasible solutions with probability models [54, 4]. Unfortunately, the ambiguity of natural languages and the inadequacy of the training process can make a semantic parser ineffective and eventually miss optimal solutions [40]. It is also worth noting that existing techniques do not attempt to select a feasible solution that maximizes or minimizes a specific metric. Although several inductive logic learning-based techniques adopt heuristic priority functions to accelerate the synthesis [46], they do not guarantee the optimality of the synthesized queries, and thus can not resolve the query candidate selection in our problem.

Our Solution. Our key idea comes from three critical observations. First, only a few relations contribute to separating positive examples from negative ones. For example, the methods in Figure 1(a) have the same modifier, indicating that the relation `Modifier` is unnecessary. Second, adding an attribute comparison expression or a string constraint to the condition of a conjunctive query yields a stronger restriction on grammatical constructs. If a query misses a positive example, we cannot obtain a query candidate by strengthening the query. Third, a desired query tends to constrain grammatical constructs mentioned in the natural language description sufficiently. For the instance in Figure 1, the query extracting the methods with the return type `CacheConfig` is a query candidate but not a desired one, as it does not pose any restriction on parameters.

Based on the observations, we realize that it is possible to narrow down necessary relations and avoid their infeasible compositions to prune the search space, and meanwhile, select query candidates with the guidance of the natural language description. According to the insight, we present a multi-modal synthesis algorithm SQUID with three stages:

- To narrow down the relations, we introduce the notion of the *dummy relations* to depict the relations unnecessary for the synthesis and propose the *representation reduction* to exclude dummy relations, which prunes the search space effectively.
- To avoid infeasible compositions of relations, we perform the *bounded refinement* to enumerate the queries, skipping the unnecessary search for the queries that exclude a positive example. Particularly, the string constraints are synthesized by computing the longest common substrings, which is achieved efficiently in the refinement.

- To select desired queries, we establish the dual quantitative metrics, namely *named entity coverage* and *structural complexity*, and select query candidates by optimizing them as the objectives, which creates more opportunities for returning desired queries.

We implement SQUID and evaluate it on 31 code search tasks. It successfully synthesizes desired queries for each task in 2.56 seconds on average. Besides, the representation reduction and the bounded refinement are crucial to its efficiency. Skipping either of them would increase the average time cost to around 8 seconds, and several tasks cannot be finished within one minute. Meanwhile, dual quantitative metrics play critical roles in the selection. Applying only one metric would make 12 or 7 tasks fail due to the synthesized non-desired queries. We also state and prove the soundness, completeness, and optimality of SQUID. If there exist query candidates for a given synthesis specification, SQUID always returns query candidates optimizing two proposed metrics to express the search intent. To summarize, our work makes the following key contributions:

- We propose a multi-modal conjunctive query synthesis problem. An effective and efficient solution can serve as a user-friendly interface of a Datalog-based analyzer for code search.
- We design an efficient algorithm SQUID for an instance of our synthesis problem, which automates the code search tasks in real-world scenarios.
- We implement SQUID as a tool and evaluate it upon 31 code search tasks, showing that SQUID synthesizes the desired queries successfully and efficiently.

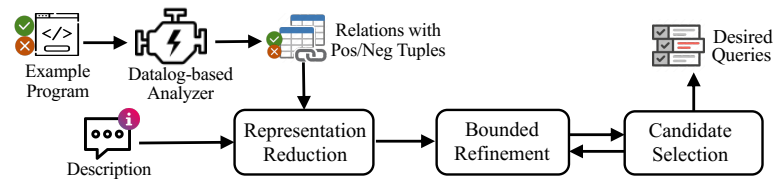
2 Overview

This section demonstrates a motivating example and briefs the key idea of our approach.

2.1 Motivating Example

Suppose a developer wants to avoid the security issue caused by `log4j` library [41]. He or she may examine the methods that receive a `Log4jUtils` object as a parameter and return a `CacheConfig` object. One choice is to leverage the built-in search tools of the IDEs to search the code lines containing `Log4jUtils` or `CacheConfig`, while the string matching-based search cannot filter grammatical constructs according to their kinds. Although several IDEs enable the structural search [23], their non-extensible templates only support searching for grammatical constructs of restrictive kinds. Another alternative is to write a query depicting the target pattern and evaluate it with a Datalog-based program analyzer, such as CODEQL [3]. However, it not only involves great laborious efforts in query language learning but also creates the burden of query writing and debugging.

To improve the usability and capability of code search, we aim to synthesize a query for a Datalog-based program analyzer. As shown in Figure 1(a), a user can specify the synthesis specification to indicate the search intent. Specifically, the positive and negative examples show the desired and undesired grammatical constructs, respectively, while the natural language description demonstrates the search intent in a sentence. Based on a Datalog-based program analyzer, we can convert the examples to a set of relations as the relational representation along with positive and negative tuples, which are shown in Figure 1(b). For example, the first tuple in the relation `Method` is the positive tuple indicating the method `foo` in Figure 1(a), which is a positive example. If we automatically synthesize the conjunctive query in Figure 1(c), the user does not need to delve into the relations and, instead specifies the synthesis specification from a high-level perspective.



■ **Figure 2** The overview of SQUID.

2.2 Synthesizing Conjunctive Queries

The query synthesizer should effectively generate the desired queries that express the search intent correctly. However, it is stunningly challenging to obtain an effective and efficient synthesizer. First, we have to tackle a great number of the relations and their attributes when we choose relevant relations and enforce correct constraints upon them, which can involve both comparisons over attributes and string constraints. Second, the non-uniqueness of query candidates creates the obstacle of selecting proper candidates. Any improper selection would return a non-desired query, leading to code search failure. To address the challenges, we propose a new multi-modal synthesis algorithm SQUID. As shown in Figure 2, SQUID consists of three phases, which come from the following three ideas.

Idea 1: Removing dummy relations. Although there are many relations potentially used in the synthesis, we can identify a class of relations, named *dummy relations*, as unnecessary ones and then discard them safely. Specifically, a relation is dummy if it cannot separate a positive tuple from a negative one. As an example, the methods in Figure 1(a) have the same modifier, which is shown by the same values of the foreign keys `mdf_id` of the relation `Method` in Figure 1(b). This indicates that the relation `Modifier` has no impact on excluding negative tuples and thus can be discarded to prune the search space. Based on this insight, we propose the **representation reduction** to remove the dummy relations, narrowing down the necessary relations for the synthesis.

Idea 2: Enumerating query candidates via refinement. According to the query syntax, the constraints, including attribute comparisons and string constraints, pose restrictions on grammatical constructs. This implies that we cannot obtain a query candidate by refining the query that excludes a positive tuple. In Figure 1, we may obtain a query that enforces both the parameter and the return value of a method have the same type. Obviously, the query excludes the method `foo`, which is a positive tuple, and thus, we should stop strengthening the restrictions on grammatical constructs. Based on this insight, we introduce the technique of the **bounded refinement**, which adds conditions for the query enumeration and discards the queries that exclude any positive tuples. Thus, we can avoid enumerating infeasible compositions of relations, which further prunes the search space effectively.

Idea 3: Dual quantitative metrics for selection. Desired queries not only separate positive tuples from negative ones but also tend to cover as many program-related named entities as possible. In Figure 1, we may obtain a query candidate that restricts the return type to be a `CacheConfig` object. However, it does not pose any restriction on the parameters and leaves the named entity “parameter” uncovered, showing that it does not express the intent sufficiently. Meanwhile, Occam’s razor [5] implies that desired queries should be as simple as possible. Hence, we introduce the *named entity coverage* and the *structural complexity*

as the dual quantitative metrics, and perform the **candidate selection** to identify desired queries by optimizing the metrics. Finally, we blend the selection with the refinement and terminate the enumeration when unexplored candidates cannot be better than the current ones, further avoiding the unnecessary enumerative search.

3 Problem Formulation

This section first presents the program relational representation (§ 3.1) and then introduces the conjunctive queries for code search (§ 3.2). Lastly, we state the multi-modal conjunctive query synthesis problem and brief the roadmap of technical sections (§ 3.3).

3.1 Program Relational Representation

First of all, we formally define the concept of the *relation* as the preliminary.

► **Definition 3.1** (Relation). A relation $R(a_1, \dots, a_n)$ is a set of tuples (t_1, \dots, t_n) , where n is the arity of R . For each $1 \leq i \leq n$, a_i is the attribute of the relation.

A relation is structured data that stores the details of different aspects of an entity. Concretely, a Datalog-based program analyzer encodes the program properties with a set of relations in a specific schema [3, 34, 37]. In what follows, we define the *relational representation* of a program.

► **Definition 3.2** (Relational Representation). Given a program, its relational representation \mathcal{R} is a set of relations over the following schema Γ . Specifically, Γ maps a relation symbol R to an n -tuple of pairs, where each element in the tuple $\Gamma(R)$ has one of the following form:

- (id, R) : The attribute id is the primary key of the relation R .
- (a, R') : The foreign key a in the relation R , referencing the primary key of $R' \in dom(\Gamma)$.
- (a, STR) : The attribute a has a string value indicating the textual information.

Particularly, we say Γ as the language schema. Without introducing the ambiguity, we use $(a, \cdot) \in \Gamma(R)$ to indicate that (a, \cdot) is an element of the tuple $\Gamma(R)$.

► **Example 3.1.** Figure 1(b) shows five relations as examples, where the values of the primary keys are italic and underlined, and the foreign keys are italic. Based on the first tuple in the relation `Method`, we can track the identifier, the return type, and the modifier of the method `foo` based on the foreign keys. Similarly, we can identify the identifier and the type of each parameter based on the relation `Parameter`.

Essentially, the relational representation of a program encodes the program properties with relations, which depicts the relationship of grammatical constructs in the program. In reality, various relations can be derived with Datalog-based program analyzers, such as `ReferenceType` and `VarPointsTo` provided by DOOP [44], depicting the type information and points-to facts, respectively. Due to the space limit, we only show five relations in Figure 1(b).

3.2 Conjunctive Queries

To simplify the presentation, we formulate the conjunctive queries as the relational algebra expressions [1] in the rest of the paper. In what follows, we first brief several relational algebra operations and then introduce the conjunctive queries for code search.

► **Definition 3.3** (Relational Algebra Operations). In a relational algebra, the selection, projection, Cartesian product, and rename operations are defined as follows:

- $\sigma_{\Theta}(R) := \{(t_1, \dots, t_n) \in R \mid [a_i \mapsto t_i \mid 1 \leq i \leq n] \models \Theta\}$ is the selection of a relation R with the selection condition Θ over its attributes.
- $\Pi_{\mathbf{a}'}(R) := \{(\mathbf{t}.a'_1, \dots, \mathbf{t}.a'_k) \mid \mathbf{t} \in R\}$ is the projection of a relation R upon a tuple of attributes \mathbf{a}' , denoted by $\Pi_{\mathbf{a}'}(R)$. Here $\mathbf{a}' = (a'_1, \dots, a'_k)$. $\mathbf{t}.a$ is the value of the attribute a in the tuple \mathbf{t} .
- $R_1 \times R_2 := \{(t_1^1, \dots, t_{n_1}^1, t_1^2, \dots, t_{n_2}^2) \mid (t_1^1, \dots, t_{n_1}^1) \in R_1, (t_1^2, \dots, t_{n_2}^2) \in R_2\}$ is the Cartesian product of two relations R_1 and R_2 .
- The rename of a relation R , denoted by $\rho_A(R)$, yields the same relation named A .

The relational algebra operations enable us to manipulate the relational representation to search desired grammatical constructs. Specifically, we often need to search specific grammatical constructs via string match and enforce them to satisfy several constraints simultaneously. Now we formalize the conjunctive queries to express the code search intent.

► **Definition 3.4** (Conjunctive Query). Given the relational representation \mathcal{R} , a conjunctive query R_Q is a relational algebra expression of the form $\Pi_{(A_i.*)}(\sigma_{\Theta}(\rho_{A_1}(R_1) \times \dots \times \rho_{A_m}(R_m)))$, where $R_i \in \mathcal{R}$, $\Theta := \phi_1 \wedge \dots \wedge \phi_n$, and each $\phi_i (1 \leq i \leq n)$ occurring in the selection condition Θ is an atomic condition in the following two forms:

- An atomic equality formula $A_j.\text{id} = A_k.a$, where a is the foreign key and $(a, R_j) \in \Gamma(R_k)$.
- A string constraint $p(A_k.a, \ell)$ over the string attribute $A_k.a$, where ℓ is a string literal, and $p \in \{\text{equal, suffix, prefix, contain}\}$.

In particular, $\Pi_{(A_i.*)}$ indicates the projection upon all the attributes of the relation A_i .

The form of the conjunctive queries in Definition 3.4 depicts the user intent from two aspects. First, the atomic equality formulas encode the relationship between the grammatical constructs. Second, the four string predicates support the common scenarios of string matching-based code search. We do not focus on synthesizing more expressive string constraints, which is the orthogonal direction of program synthesis [10, 27, 36]. Based on the conjunctive queries, we can simultaneously perform the string matching-based search and filter the constructs with various relations in the program relational representation.

► **Example 3.2.** We can formalize the query in Figure 1(c) as the relational algebra expression:

$$\Pi_{(A_1.*)}(\sigma_{\Theta}(\rho_{A_1}(\text{Method}) \times \rho_{A_2}(\text{Type}) \times \rho_{A_3}(\text{Parameter}) \times \rho_{A_4}(\text{Type})))$$

where the selection condition Θ in the selection operation is as follows:

$$\Theta := (A_1.\text{id} = A_3.\text{method_id}) \wedge (A_1.\text{ret_type_id} = A_2.\text{id}) \wedge (A_3.\text{type_id} = A_4.\text{id}) \wedge \\ \text{equal}(A_2.\text{name}, \text{“CacheConfig”}) \wedge \text{equal}(A_4.\text{name}, \text{“Log4jUtils”})$$

The conjunctive queries can be instantiated with various flavors [17, 1, 7]. The *select-from-where queries* are the instantiations of the conjunctive queries in SQL. Besides, a simple Datalog program can also express the conjunctive query with a single Datalog rule. In our paper, we formulate a conjunctive query as a relational algebra expression. Our implementation actually synthesizes the conjunctive queries as Datalog programs, which are evaluated by a Datalog solver over the program relational representation for code search.

3.3 Multi-modal Conjunctive Query Synthesis Problem

To generate the conjunctive query for code search, the users need to specify their intent as the specification. In our work, we follow the premise of the recent studies on the multi-modal program synthesis [10, 4] that multiple modalities of information can go arm in arm with each other, serving as the informative specification for the synthesis. Specifically, the users provide a natural language sentence to describe the target code pattern and provide several grammatical constructs as positive and negative examples. Notably, the multi-modal synthesis specification is easy to provide. When the users want to explore a specific code pattern, they can describe the pattern briefly in a natural language and provide several examples from their editors instead of delving into the details of the underlying relations and their attributes, enabling users to generate a query for code search in a declarative manner.

Based on the multi-modal synthesis specification, the positive and negative examples are converted to the tuples in a specific relation. For example, Figure 1 shows a set of relations as the relational representation of the examples. To formalize our problem better, we define the notion of the *relation partition* as follows.

► **Definition 3.5 (Relation Partition).** The relation partition of $R^* \in \mathcal{R}$ is a pair of two relations (R_p^*, R_n^*) satisfying $R^* = R_p^* \cup R_n^*$ and $R_p^* \cap R_n^* = \emptyset$. The relation partition is non-trivial if and only if $R_p^* \neq \emptyset$ and $R_n^* \neq \emptyset$. We say the tuples in R_p^* and R_n^* are positive and negative tuples, respectively.

► **Example 3.3.** As shown in Figure 1, we can construct the relation partition (R_p^*, R_n^*) , where $R_p^* = \{(M1, I1, T3, MDF1)\}$ and $R_n^* = \{(M2, I2, T3, MDF1), (M3, I3, T2, MDF1)\}$. Obviously, R_p^* and R_n^* are disjoint, and $R_p^* \cup R_n^*$ is exactly the relation Method.

The positive and negative tuples essentially depict the positive and negative examples, respectively. Based on the program’s relational representation, the examples specified by the users can determine the positive and negative tuples, which can be achieved in various manners. Specifically, users can select a grammatical construct in the IDEs or use a code sample in a specific coding standard [51] as a positive example and remove several sub-patterns from a positive example by mutation to construct negative ones. Such positive and negative examples further constitute a sample code snippet, from which a Datalog-based analyzer derives a set of relations as the relational representation. In this paper, we omit the details of positive/negative tuple generation and formulate a *multi-modal conjunctive query synthesis (MMCQS) problem* as follows.

Given a relational representation \mathcal{R} , a relation partition (R_p^*, R_n^*) of $R^* \in \mathcal{R}$, and a natural language description s , we aim to synthesize a conjunctive query R_Q containing the positive tuples in R_p^* and excluding the negative tuples in R_n^* .

► **Example 3.4.** Figure 1(a) shows the multi-modal synthesis specification, which consists of a positive example, two negative examples, and a natural language description s as “Find all the methods receiving a `Log4jUtils`-type parameter and giving a `CacheConfig`-type return”. Leveraging a Datalog-based analyzer, we obtain the relational representation and the relation partition of Method, which are shown in Figure 1(b). To automate the code search, we expect to synthesize the query in Fig 1(c) or Example 3.2 according to the relational representation, the relation partition, and the natural language sentence.

To promote the code search, we propose an efficient synthesis algorithm SQUID for the MMCQS problem, which is our main technical contribution. As explained in § 1, it is challenging to solve the MMCQS problem efficiently, which involves mitigating the huge

search space and selecting queries from multiple candidates. In the following two sections, we formalize the conjunctive query synthesis from a graph perspective (§ 4), and illustrate the technical details of our synthesis algorithm SQUID (§ 5), which prunes the search space and selects desired queries effectively.

4 Conjunctive Query Synthesis: A Graph Perspective

This section presents a graph perspective of our conjunctive query synthesis problem. Specifically, we introduce two graph representations of the language schema and the conjunctive queries, named the schema graph (§ 4.1) and the query graph (§ 4.2), respectively, which reduces the conjunctive query synthesis to the query graph enumeration. Lastly, we summarize the section and highlight the technical challenges from a graph perspective (§ 4.3).

4.1 Schema Graph

According to Definition 3.2, a relation in the language schema has three kinds of attributes, namely a unique primary key, foreign keys, and string attributes. Obviously, the selection condition Θ in R_Q should only compare the foreign key of a relation with its referenced primary key or constrain the string attributes with string predicates. To depict the possible ways of constraining the attributes, we define the concept of the *schema graph* as follows.

► **Definition 4.1** (Schema Graph). The schema graph G_Γ of a language schema Γ is (N_Γ, E_Γ) :

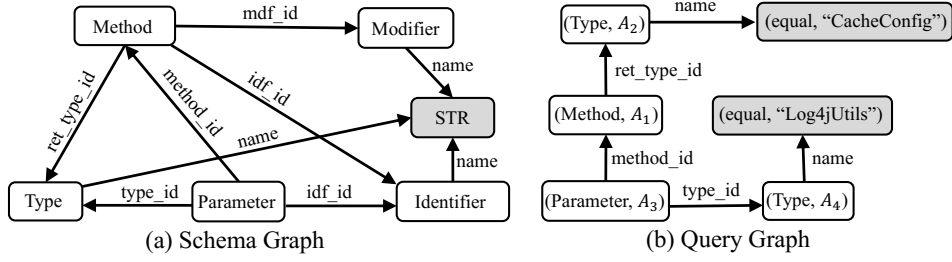
- The set N_Γ contains the relation symbols in the schema or the string type STR as the nodes of the schema graph, i.e., $N_\Gamma := \text{dom}(\Gamma) \cup \{\text{STR}\}$.
- The set E_Γ contains an edge (n_1, n_2, a) if and only if either of the conditions holds:
 - $n_1, n_2 \in \text{dom}(\Gamma)$ and $(a, n_2) \in \Gamma(n_1)$: The relation n_1 has a foreign key named a referencing the primary key of the relation n_2 .
 - $n_1 \in \text{dom}(\Gamma)$ and $(a, \text{STR}) \in \Gamma(n_1)$: a is the string attribute of the relation n_1 .

► **Example 4.1.** Consider the relations in the relational representation shown in Figure 1(b). We can construct the schema graph in Figure 3(a). The edge from Method to Modifier labeled with `mdf_id` shows that the attribute `mdf_id` of Method is a foreign key referencing the primary key of Modifier. Similarly, the edge from Type to STR labeled with `name` shows that the attribute `name` in Type is a string attribute.

Noting that there can exist multiple edges with different labels between two nodes in the schema graph, which indicate that a relation take multiple foreign keys referencing the same relation or string attributes as the attributes. Essentially, the schema graph encodes the available relations with its node set and depicts the valid forms of the atomic formulas appearing in the selection condition with its edge set. Although we can compare the attributes of any relations flexibly, a solution to our problem must take the valid form of the atomic formulas as its selection condition, comparing the foreign keys with the referenced primary keys or examining the string attributes of the relations appearing in a Cartesian product.

4.2 Query Graph

As formulated in Definition 3.4, there are two key components in the conjunctive query, namely the Cartesian product and the selection condition. Leveraging the schema graph, we can represent the components with nodes and edges on the graph, which uniquely determines a conjunctive query. Formally, we introduce the notion of the *query graph* as follows.



■ **Figure 3** The examples of the schema graph and the query graph.

► **Definition 4.2** (Query Graph). Given a conjunctive query Q , its query graph G_Q is (N_Q, E_Q, Φ_Q) :

- The set N_Q contains (R_i, A_i) or (p, ℓ) as a node in the query graph. $R_i \in \mathcal{R}$ is a relation and A_i is the unique relation identifier. p and ℓ are the string predicate and literal, respectively.
- The set E_Q contains (n_1, n_2, a) as an edge, corresponding to the equality atomic formula $A_j.a = A_k.id$ in the selection condition, where $n_1 = (R_j, A_j)$ and $n_2 = (R_k, A_k)$.
- The mapping Φ_Q maps a 3-tuple (R_j, A_j, a) to a node (p, ℓ) , indicating the edge from (R_j, A_j) to (p, ℓ) with the label a , which corresponds to the string constraint $p(A_j.a, \ell)$.

► **Example 4.2.** Figure 3(b) shows an query graph of R_Q in Example 3.2. The white nodes show the four relations appearing in the Cartesian product, while the gray nodes indicate the string predicates and literals in the selection condition. The edges depict two kinds of atomic conditions. For example, the edge from $(Parameter, A_3)$ to $(Method, A_1)$ labeled with `method_id` indicates the equality constraint $A_3.method_id = A_1.id$. Meanwhile, the edge induced by $\Phi_Q(\text{Type}, A_4, \text{name}) = (\text{equal}, \text{"Log4jUtils"})$ indicates the string constraint $\text{equal}(A_4.name, \text{"Log4jUtils"})$.

Essentially, a conjunctive query and a query graph are allotropes. That is, there exists a bijection κ mapping a conjunctive query R_Q to a query graph G_Q such that $G_Q = \kappa(R_Q)$ and $R_Q = \kappa^{-1}(G_Q)$. Thus, we can enumerate the conjunctive queries by enumerating the query graphs. Besides, the schema graph restricts the form of the selection condition over the attributes. If an edge with the label a connects (R_j, A_j) and (R_k, A_k) in a query graph, there should exist an edge labeled with a connecting the relations R_j and R_k in the schema graph. A similar argument also holds for the edge of the query graph indicated by the mapping Φ_Q . Therefore, we can reduce our conjunctive query synthesis to a search problem, of which the search space is characterized as the set of query graphs.

4.3 Summary

Leveraging the schema graph, we have reduced the conjunctive query synthesis process to query graph enumeration. To obtain the desired queries, we only need to select the nodes and edges from the schema graph and create the node (p, ℓ) with proper string predicates and literals for constructing a query graph G_Q , which should satisfy that the induced query $\kappa^{-1}(G_Q)$ separates positive tuples from negative ones.

Obtaining the desired queries with high efficiency is a non-trivial problem. The schema graph can be overwhelming, containing over a hundred nodes and edges, which leads to an enormous number of choices for relations occurring in the query graph. Even for a given set of

relations, the flexibility of instantiating equality constraints and string constraints over their attributes can induce a large number of selection choices of edges in a query graph, which exacerbates the search space explosion problem. Additionally, the existence of multiple query candidates necessitates the effective selection of candidates, which is crucial for conducting a code search task. In the next section, we will detail our synthesis algorithm that addresses these challenges, resulting in high efficiency and effectiveness for code search.

5 Synthesis Algorithm

This section presents our synthesis algorithm SQUID to solve the MMCQS problem. SQUID takes as input the relational representation \mathcal{R} of an example program, a relation partition (R_p^*, R_n^*) of $R^* \in \mathcal{R}$, and a natural language description s . It generates the query candidates separating positive tuples $\mathbf{t}_p \in R_p^*$ from negative ones $\mathbf{t}_n \in R_n^*$, which can be further selected and then used for code search. To address the challenges in § 4.3, SQUID works with the following three stages:

- To tackle a large number of relations, SQUID relies on the notion of dummy relations and conducts the representation reduction based on the positive and negative tuples, which effectively narrow down the relations that can appear in the query graph (§ 5.1).
- To avoid unnecessary enumeration of edges in query graphs, we propose the bounded refinement by enumerating the query graphs based on the schema graph, which essentially appends equality constraints and string constraints inductively (§ 5.2).
- To select query candidates, SQUID identifies the named entities in the natural language description s for the prioritization, and blends the selection with the refinement to collect the desired queries (§ 5.3).

We also formulate the soundness, completeness, and optimality of our algorithm (§ 5.4). For better illustration, we use the synthesis instance shown in Figure 1 throughout this section.

5.1 Representation Reduction

To tackle the large search space, we first propose the representation reduction to narrow down the relations possibly used in the query. Specifically, we introduce the notion of the dummy relations to determine the characteristics of unnecessary relations (§ 5.1.1) and propose the algorithm of removing dummy relations for the representation reduction (§ 5.1.2).

5.1.1 Dummy Relations

There exists a class of relations, named *dummy relations*, which cannot involve in distinguishing positive and negative tuples, such as the relation **Modifier** in Figure 1. Before defining them, we first introduce the *undirected relation path* and the *activated relation*.

► **Definition 5.1** (Undirected Relation Path). An undirected relation path from R_0 to R_{k+1} in the schema graph $G_\Gamma = (N_\Gamma, E_\Gamma)$ is $p : R_0 \xleftrightarrow{(a_0, d_0)} \cdots \xleftrightarrow{(a_k, d_k)} R_{k+1}$, where $R_i \in \text{dom}(\Gamma)$. Here, $d_i = 1$ if and only if $(R_i, R_{i+1}, a_i) \in E_\Gamma$, and $d_i = -1$ if and only if $(R_{i+1}, R_i, a_i) \in E_\Gamma$.

► **Definition 5.2** (Activated Relation). Given a tuple $\mathbf{t}_0 \in R_0$ and an undirected relation path $p : R_0 \xleftrightarrow{(a_0, d_0)} \cdots \xleftrightarrow{(a_k, d_k)} R_{k+1}$, the activated relation of \mathbf{t}_0 along p is

$$\mathcal{I}(\mathbf{t}_0, p) = \{\mathbf{t}_{k+1} \mid \mathbf{t}_{i+1} \in R_{i+1}, \text{ite}(d_i = 1, \mathbf{t}_i.a_i = \mathbf{t}_{i+1}.\text{id}, \mathbf{t}_i.\text{id} = \mathbf{t}_{i+1}.a_i), 0 \leq i \leq k\}$$

► **Example 5.1.** In Figure 1, the path $p_1 : \text{Method} \xleftrightarrow{(\text{method_id}, -1)} \text{Parameter} \xleftrightarrow{(\text{type_id}, 1)} \text{Type}$, and $\mathbf{t}_p = (\text{M1}, \text{l1}, \text{T3}, \text{MDF1}) \in R_p^*$. We have $\mathbf{t}_1 = (\text{P1}, \text{l4}, \text{T1}, \text{M1}) \in \text{Parameter}$, and $\mathbf{t}_2 = (\text{T1}, \text{Log4jUtils}) \in \text{Type}$. By inspecting other tuples, we have $\mathcal{I}(\mathbf{t}_p, p_1) = \{(\text{T1}, \text{Log4jUtils})\}$.

36:12 Synthesizing Conjunctive Queries for Code Search

Intuitively, an undirected relation path can depict the restriction upon the relations in the Cartesian product of the query. The activated relation actually contains the tuples enforcing the primary key of \mathbf{t}_0 to appear in a selected tuple. Therefore, it is possible to narrow down the relations for the synthesis by inspecting the activated relations of positive and negative tuples along each undirected relation path. According to the intuition, we formally introduce and define the notion of the *dummy relations* as follows.

► **Definition 5.3** (Dummy Relation). Given a relation partition (R_p^*, R_n^*) of $R^* \in \mathcal{R}$, a relation $R \in \mathcal{R}$ is dummy if for every undirected relation path p from R^* to R , either of the conditions is satisfied: (1) There exists $\mathbf{t}_p \in R_p^*$ such that $\mathcal{I}(\mathbf{t}_p, p) = \emptyset$; (2) $\mathcal{I}(\mathbf{t}_p, p) = \mathcal{I}(\mathbf{t}_n, p)$ for any $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$.

Definition 5.3 formalizes two characteristics of relations unnecessary the synthesis. First, the empty activated relation of a positive tuple \mathbf{t}_p indicates the absence of the tuples in the relation R , making the tuple \mathbf{t}_p appear. Second, the relation R cannot contribute to separating positive tuples from negative ones if several tuples in R make all the positive and negative tuples appear simultaneously. Thus, such relations can be discarded safely.

► **Example 5.2.** Example 5.1 shows that $\mathcal{I}(\mathbf{t}_p, p_1) \neq \emptyset$. Similarly, $\mathcal{I}(\mathbf{t}_n, p_1) = \{(\text{T2}, \text{int})\} \neq \mathcal{I}(\mathbf{t}_p, p_1)$ when $\mathbf{t}_n = (\text{M2}, \text{l2}, \text{T4}, \text{MDF1}) \in R_n^*$. Hence, `Type` is not dummy. Besides, any undirected relation path p_2 from `Method` to `Modifier` has the form

$$\text{Method}[\xrightarrow{(\text{mdf_id}, +1)} \text{Modifier} \xrightarrow{(\text{mdf_id}, -1)} \text{Method}]^* \xrightarrow{(\text{mdf_id}, +1)} \text{Modifier}$$

where $[\cdot]^*$ indicates the repeated subpath. We find that $\mathcal{I}(\mathbf{t}_p, p_2) = \mathcal{I}(\mathbf{t}_n, p_2) = \{\text{MDF1}, \text{public}\}$ for any $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$. Thus, the relation `Modifier` is a dummy relation.

5.1.2 Removing Dummy Relations

Based on Definition 5.3, identifying dummy relations involves two technical parts. First, we should collect all the undirected relation paths from R^* to each relation. Second, we need to compute $\mathcal{I}(\mathbf{t}_p, p)$ and $\mathcal{I}(\mathbf{t}_n, p)$ for each undirected relation path p and positive/negative tuple. However, the schema graph can contain a large and even infinite number of undirected relation paths from R^* . Any cycle induces the infinity of the path number, making it tricky to examine the conditions in Definition 5.3. Fortunately, we realize that the number of cycles in a path does not affect the activated relation, which is stated in the following property.

► **Property 5.1.** Given any $\mathbf{t}_0 \in R_0$, we have $\mathcal{I}(\mathbf{t}_0, p) = \mathcal{I}(\mathbf{t}_0, p')$ for p and p' as follows:

$$\begin{aligned} p &: R_0 \xrightarrow{(a_0, d_0)} \cdots R_l [\xrightarrow{(a'_l, d'_l)} \cdots R_{l+t} \xrightarrow{(a'_{l+t}, d'_{l+t})} R_l]^+ \xrightarrow{(a_l, d_l)} \cdots R_k \xrightarrow{(a_k, d_k)} R_{k+1} \\ p' &: R_0 \xrightarrow{(a_0, d_0)} \cdots R_l \xrightarrow{(a'_l, d'_l)} \cdots R_{l+t} \xrightarrow{(a'_{l+t}, d'_{l+t})} R_l \xrightarrow{(a_l, d_l)} \cdots R_k \xrightarrow{(a_k, d_k)} R_{k+1} \end{aligned}$$

Here, $[\cdot]^+$ indicates the cycle occurring at least one time.

Property 5.1 holds trivially according to Definition 5.2. For each undirected relation path p containing a cycle, the constraints over the tuples in $\mathcal{I}(\mathbf{t}_0, p)$ are the same as the ones over the tuples in $\mathcal{I}(\mathbf{t}_0, p')$. Thus, we can just examine the finite number of undirected relation paths in which a cycle appears at most one time.

■ **Algorithm 1** Removing dummy relations for representation reduction.

```

1 Procedure reduce( $\Gamma, \mathcal{R}, R_p^*, R_n^*$ ):
2    $\mathcal{R}' \leftarrow \emptyset$ ;
3    $G_\Gamma \leftarrow \text{SchemaGraph}(\Gamma)$ ;
4   foreach  $R \in \mathcal{R}$  :
5      $\mathcal{P} \leftarrow \text{augmentPathWithCycle}(\text{AcyclicPath}(R^*, R, G_\Gamma))$ ;
6     foreach  $p \in \mathcal{P}, \mathbf{t}_p \in R_p^*, \mathbf{t}_n \in R_n^*$  :
7       if  $\mathcal{I}(\mathbf{t}_p, p) \neq \mathcal{I}(\mathbf{t}_n, p)$  :
8          $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$ ; break ;
9     foreach  $p \in \mathcal{P}, \mathbf{t}_p \in R_p^*$  :
10      if  $\mathcal{I}(\mathbf{t}_p, p) = \emptyset$  :
11         $\mathcal{R}' \leftarrow \mathcal{R}' \setminus \{R\}$ ; break ;
12  return  $\mathcal{R}'$ ;

```

Establish upon the above concepts and property, Algorithm 1 shows the details of the representation reduction by removing dummy relations. Initially, we construct the schema graph G_Γ according to Definition 4.1 (line 3). Then we compute the undirected relation paths from R^* to R in G_Γ , where any cycle repeats at most once (lines 4–5). Specifically, the function `AcyclicPath` collects all the acyclic undirected relation paths from R^* to R in the schema graph G_Γ , while the function `augmentPathWithCycle` augments each acyclic path by appending each cycle at most one time. For each undirected relation path p , we compute $\mathcal{I}(\mathbf{t}_p, p)$ and $\mathcal{I}(\mathbf{t}_n, p)$ according to Definition 5.2 for each $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$, respectively. Therefore, we can identify R as a non-dummy relation if both the conditions in Definition 5.3 are violated (lines 6–11). Finally, we obtain the reduced relational representation \mathcal{R}' that excludes all the dummy relations (line 12).

► **Example 5.3.** Consider the undirected relation paths from Method to Modifier in Figure 3(a). Algorithm 1 collects the acyclic path $p_3 : \text{Method} \xrightarrow{(\text{mdf_id}, +1)} \text{Modifier}$ and augments it to form the path $p_4 : \text{Method} \xrightarrow{(\text{mdf_id}, +1)} \text{Modifier} \xrightarrow{(\text{mdf_id}, -1)} \text{Method} \xrightarrow{(\text{mdf_id}, +1)} \text{Modifier}$. Based on the activated relations $\mathcal{I}(\mathbf{t}, p_3)$ and $\mathcal{I}(\mathbf{t}, p_4)$ for each tuple \mathbf{t} in Method, we can find that $\mathcal{I}(\mathbf{t}_p, p) = \mathcal{I}(\mathbf{t}_n, p)$ for every $\mathbf{t}_p \in R_p^*$ and $\mathbf{t}_n \in R_n^*$. Thus, Modifier is a dummy relation.

Essentially, our representation reduction analyzes the example program upon its relational representation. The activated relations provide sufficient clues to identifying unnecessary relations, i.e., the dummy ones. As the first step of the synthesis, the representation reduction narrows down the relations used in the conjunctive query. Furthermore, in the enumeration of the edges of a query graph, i.e., the sets E_Q and Φ_Q , we only need to focus on the attributes in the non-dummy relations in the reduced relational representation, which prunes the search space significantly. Lastly, we formulate the soundness of the representation reduction as follows, which can further ensure the completeness of our synthesis algorithm in § 5.4. We provide a detailed proof in [49].

► **Theorem 5.1.** (Soundness of Representation Reduction) If an instance of the MMCQS problem has a solution, there must be a conjunctive query R_Q , of which the Cartesian product only consists of non-dummy relations, such that R_Q is also a solution.

5.2 Bounded Refinement

Based on the reduced relational representation \mathcal{R}' , we can enumerate query candidates by selecting proper nodes corresponding to non-dummy relations in \mathcal{R}' and edges connecting such nodes. However, the search space is potentially unbounded. The relations can occur in

a query multiple times, i.e., a node in the schema graph can be selected more than one time. Meanwhile, the literal in a string constraint can be instantiated flexibly, which increases the difficulty of enumerating a query graph with proper instantiation of Φ_Q . To achieve high efficiency, we propose the bounded refinement to expand query graphs on demand and strengthen the query with the strongest string constraints. Specifically, we first introduce the notions of the bounded query and the refinable query (§ 5.2.1), and then present the details of enumerating the query graphs (§ 5.2.2).

5.2.1 Bounded Query and Refinable Query

As shown in Example 3.2, a relation can appear multiple times in the Cartesian product of a conjunctive query, inducing an unbounded search space in the synthesis. The unboundedness of the search space poses the great challenge of enumerating the query candidates efficiently. However, we realize that the conjunctive query for a code search task often involves only a few relations, each of which appears quite a few times. Thus, it is feasible to bound the maximal multiplicity of the relation in the query and conduct the bounded enumeration. Formally, we introduce the notion of the (m, k) -bounded query as follows.

► **Definition 5.4** ((m, k) -Bounded Query). An (m, k) -bounded query is a conjunctive query with m relations such that (1) each relation appears at most k times; (2) there is a relation appearing exactly k times.

► **Example 5.4.** The conjunctive query $\Pi_{(A_1.*)}(\sigma_{\text{true}}(\rho_{A_1}(\text{Method})))$ is a $(1, 1)$ -bounded query. Similarly, the conjunctive query in Example 3.2 is a $(4, 2)$ -bounded query.

Intuitively, we can enumerate the (m, k) -bounded queries by selecting non-dummy relations at most k times, forming a query graph with m nodes. When constructing the sets E_Q and Φ_Q for the query graph enumeration, we only need to concentration on the attributes of the selected relations. However, not all the query graphs are worth enumerating. If R_Q excludes a positive tuple, there is no need to add more nodes and edges to its query graph, as it would induce a stronger selection condition, making the new query still exclude the positive tuple. Formally, we formulate the notion of the *refinable query* as follows.

► **Definition 5.5** (Refinable Query). A conjunctive query R_Q is a refinable query if and only if for any $\mathbf{t}_p \in R_p^*$ we have $\mathbf{t}_p \in R_Q$, i.e., $R_p^* \subseteq R_Q$.

► **Example 5.5.** Consider $R_Q := \Pi_{(A_1.*)}(\sigma_{\Theta}(\rho_{A_1}(\text{Method}) \times \rho_{A_2}(\text{Type}) \times \rho_{A_3}(\text{Parameter})))$, where the selection condition Θ in the selection operation is as follows:

$$\Theta := (A_1.\text{id} = A_3.\text{method_id}) \wedge (A_1.\text{ret_type_id} = A_2.\text{id}) \wedge \text{equal}(A_2.\text{name}, \text{"CacheConfig"})$$

It is a refinable query as $R_p^* \subseteq R_Q = \{(M1, I1, T3, MDF1), (M2, I2, T3, MDF1)\}$.

Essentially, a refinable query is the over-approximation of the positive tuples. When it excludes all the negative tuples, the query is exactly a query candidate. Thus, we can collect the query candidates by refining the refinable queries in the bounded enumeration.

5.2.2 Enumerating Query Candidates via Refinement

We denote the sets of (m, k) -bounded refinable queries and query candidates by $\mathcal{S}_R(m, k)$ and $\mathcal{S}_C(m, k)$, respectively, and set a multiplicity bound K to bound the multiplicity of a relation. To conduct a bounded enumeration, we have to compute the set $\mathcal{S}_C(m, k)$ for $k \leq K$,

■ **Algorithm 2** Enumerating query candidates via refinement.

```

1 Procedure refine( $\mathcal{S}_R, \mathcal{S}_C, R_p^*, R_n^*, m, k, \mathcal{R}'$ ):
2   if  $m = 1$  and  $k = 1$  :
3      $W \leftarrow \{(\emptyset, \emptyset, \perp)\}$ ;
4   if  $m > 1$  :
5      $W \leftarrow \{\kappa(R_Q) \mid R_Q \in \mathcal{S}_R(m-1, k)\}$ ;
6     if  $k > 1$  :
7        $W \leftarrow W \cup \{\kappa(R_Q) \mid R_Q \in \mathcal{S}_R(m-1, k-1)\}$ ;
8
9    $\mathcal{S}_R(m, k) \leftarrow \emptyset$ ;
10  while  $W$  is not empty do
11     $G_Q \leftarrow \text{pop}(W)$ ;  $V \leftarrow \emptyset$ ;
12    foreach  $R \in \mathcal{R}'$  :
13      if  $\text{multiplicity}(G_Q, R) < k$  and  $\kappa^{-1}(G_Q) \in \mathcal{S}_R(m-1, k)$  :
14         $V \leftarrow V \cup \text{expand}(G_Q, R)$ ;
15      if  $\text{multiplicity}(G_Q, R) = (k-1)$  and  $\kappa^{-1}(G_Q) \in \mathcal{S}_R(m-1, k-1)$  :
16         $V \leftarrow V \cup \text{expand}(G_Q, R)$ ;
17
18    foreach  $G_Q : (N_Q, E_Q, \Phi_Q) \in V$  and  $R_p^* \subseteq \kappa^{-1}(G_Q)$  :
19       $\mathcal{S}_R(m, k) \leftarrow \mathcal{S}_R(m, k) \cup \kappa^{-1}(G_Q)$  ;
20      foreach  $(R_i, A_i) \in N_Q$  and  $(a, STR) \in \Gamma(R_i)$  :
21         $(p, \ell) \leftarrow \text{synLCS}(G_Q, R_i, A_i, a, R_p^*)$ ;
22         $G'_Q \leftarrow (N_Q, E_Q, \Phi_Q[(R_i, A_i, a) \mapsto (p, \ell)])$  ;
23         $\mathcal{S}_R(m, k) \leftarrow \mathcal{S}_R(m, k) \cup \kappa^{-1}(G'_Q)$ ;
24
25   $\mathcal{S}_C(m, k) \leftarrow \{R_Q \mid R_Q \in \mathcal{S}_R(m, k), R_p^* = R_Q\}$ ;

```

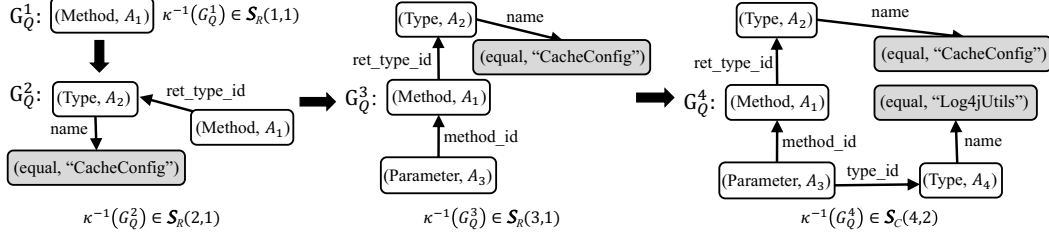
which can be achieved by examining whether the queries in $\mathcal{S}_R(m, k)$ are query candidates. Obviously, exhaustive enumeration is impossible as the search space of (m, k) -bounded queries is exponential to m and k . To avoid unnecessary enumeration, we leverage the structure of $\mathcal{S}_C(m, k)$, which is formulated in the following property.

► **Property 5.2.** For every refinable query $R_Q \in \mathcal{S}_C(m, k)$ and $\kappa(R_Q) := (N_Q, E_Q, \Phi_Q)$, there exists a query graph G_Q^1 or G_Q^2 such that

- $N_Q = N_Q^1 \cup \{(R, A_i)\}$, $E_Q^1 \subseteq E_Q$, and $\Phi_Q^1 \subseteq \Phi_Q$, where R appears in G_Q^1 exactly $(k-1)$ times. Here, $G_Q^1 = (N_Q^1, E_Q^1, \Phi_Q^1)$ and $\kappa^{-1}(G_Q^1) \in \mathcal{S}_R(m-1, k-1)$.
- $N_Q = N_Q^2 \cup \{(R, A_i)\}$, $E_Q^2 \subseteq E_Q$, and $\Phi_Q^2 \subseteq \Phi_Q$, where R appears in G_Q^2 fewer than k times. Here, $G_Q^2 = (N_Q^2, E_Q^2, \Phi_Q^2)$ and $\kappa^{-1}(G_Q^2) \in \mathcal{S}_R(m-1, k)$.

Property 5.2 shows that the sets of the nodes and edges in a query graph of a refinable query are subsumed by the ones of a refinable query with fewer relations, which permit us to enumerate the query candidates by computing $\mathcal{S}_R(m, k)$ and $\mathcal{S}_C(m, k)$ inductively. Technically, we achieve the enumerative search via the bounded refinement. Assuming that we have $\mathcal{S}_R(m', k')$ and $\mathcal{S}_C(m', k')$ for all $m' < m$ and $k' \leq k$. Algorithm 2 computes the sets $\mathcal{S}_R(m, k)$ and $\mathcal{S}_C(m, k)$. The technical details of the refinement are as follows:

- For the base case, where $m = 1$ and $k = 1$, we construct an empty query graph (line 3). For a general case, we merge the sets of the query graphs induced by the refinable queries in $\mathcal{S}_R(m-1, k)$ and $\mathcal{S}_R(m-1, k-1)$ (lines 4-7). Hence, we obtain a set of query graphs W to maintain all the refinable queries with m relations.



■ **Figure 4** The example of the bounded refinement.

- For each query graph $G_Q \in W$, we leverage the function `expand` wraps a specific relation R as a new node and add new edges, producing a set of query graphs containing the relation R (lines 12–16). Such query graphs are maintained in the set V .
- For each query graph $G_Q \in V$ that induces a refinable query, we add the induced refinable query to the set $\mathcal{S}_R(m, k)$ (line 19) and attempt to synthesize a string constraint (lines 20–23). Specifically, the function `synLCS` examines the values of a string attribute a in the positive tuples to compute the longest common substring ℓ and the strongest string predicate p (line 21), where all the values of a in the positive tuples satisfy the string constraints induced by p and ℓ . By updating Φ_Q , we add a new edge to G_Q and produce a new query graph inducing a refinable query (lines 22–23).
- Finally, we check whether a (m, k) -bounded refinable query is a query candidate or not, which yields the set $\mathcal{S}_C(m, k)$ (line 25). The sets $\mathcal{S}_R(m, k)$ and $\mathcal{S}_C(m, k)$ are exactly the sets of (m, k) -bounded refinable queries and query candidates.

Notably, we construct the string constraints on demands to strengthen the selection condition of refinable queries, which is achieved by the function `synLCS` at line 21. The reduction from string constraint synthesis to the LCS computation enable us to leverage existing algorithms, such as general suffix automaton [33], to compute the string literal ℓ and select a string predicate p efficiently, which promote the efficiency of query refinement.

► **Example 5.6.** Figure 4 shows the part of the refinement for the instance in Example 3.4. G_Q^1 only contains the relation `Method`. After adding the nodes and edges, we construct the query graphs of refinable queries with more relations and atomic constraints, such as G_Q^2 , G_Q^3 , and G_Q^4 . Particularly, we identify the query candidate $\kappa^{-1}(G_Q^4)$, i.e., R_Q in Example 3.2.

5.3 Candidate Selection

Based on the bounded refinement, we collect the query candidates in which each relation appears no more than K times, where K is the multiplicity bound. However, not all the query candidates are the desired ones. In this section, we introduce dual quantitative metrics to prioritize queries (§ 5.3.1) and select query candidates during the refinement (§ 5.3.2).

5.3.1 Dual Quantitative Metrics

Desired queries are expected to express the search intent correctly, covering as many grammatical concepts in the natural language as possible. Also, they should be as simple as possible according to Occam’s razor. Based on the intuitions, we formalize the following two metrics and then define the total order to prioritize the query candidates.

► **Definition 5.6.** (Named-Entity Coverage) Given a function h mapping an attribute of a relation to a set of words in natural language, the named entity coverage of a conjunctive query R_Q with respect to a natural language description s is

$$\alpha_h(R_Q, s) = \frac{1}{|N(s)|} \cdot \left| \bigcup_{(R_i, a_j) \in \mathcal{A}(\Theta)} h(R_i, a_j) \cap N(s) \right|$$

Here, $\mathcal{A}(\Theta)$ contains the relations and their attributes appearing in the selection condition Θ while $N(s)$ is the set of the named entities in the description s .

► **Definition 5.7.** (Structural Complexity) Let $\delta(\Theta)$ be the number of the atomic formulas in the selection condition Θ . The structural complexity of a conjunctive query R_Q is

$$\beta(Q) = m + \delta(\Theta)$$

To compute the named entity coverage, we instantiate the function h manually and obtain the set $N(s)$ from the natural language description s based on the named entity recognition techniques [31]. The computation does not introduce much overhead, as the natural language description exhibits a fairly small length in our scenarios. Meanwhile, measuring the structural complexity is quite straightforward. The quantities m and $\delta(\Theta)$ can be totally determined by the sizes of the sets N_Q , E_Q and Φ_Q in a query graph. Thus, computing the two quantities does not introduce much overhead during the enumeration.

► **Example 5.7.** Assume that we instantiate the function h as follows:

$$\begin{aligned} h(\text{Parameter}, \text{id}) &= \{\text{parameter}\}, \quad h(\text{Method}, \text{id}) = \{\text{method}\}, \quad h(\text{Method}, \text{idf_id}) = \{\text{identifier}\} \\ h(\text{Method}, \text{ret_type_id}) &= \{\text{return}, \text{type}\}, \quad h(\text{Method}, \text{mdf_id}) = \{\text{modifier}\} \end{aligned}$$

Consider the natural language description s in Example 3.4 and the conjunctive query R_Q in Example 3.2, of which the query graph is shown in Figure 3(b). We can obtain a set of named entities $N(s) = \{\text{method}, \text{type}, \text{parameter}, \text{return}\}$. Therefore, we have $\alpha_h(R_Q, s) = 1$. According to $m = 4$ and $\delta(\Theta) = 5$, its structural complexity is $\beta(R_Q) = 4 + \delta(\Theta) = 9$.

Intuitively, the selection condition of a query is more likely to conform to the user intent if the query has a higher named entity coverage. Besides, the simpler form query can have better generalization power among the query candidates covering the same number of named entities. Based on Occam's razor, we should choose the simplest query from the candidates that maximizes the named entity coverage. Thus, we propose the *total order* of conjunctive queries as follows.

► **Definition 5.8** (Total Order). Given the function h in Definition 5.6, we have $R_Q^2 \preceq_s R_Q^1$ if and only if they satisfy one of the following conditions:

$$\begin{aligned} \alpha_h(R_Q^1, s) &\geq \alpha_h(R_Q^2, s) \quad \text{or} \\ \alpha_h(R_Q^1, s) &= \alpha_h(R_Q^2, s), \quad \beta(R_Q^1) \leq \beta(R_Q^2) \end{aligned}$$

► **Example 5.8.** Consider the following query candidates for the instance in Example 3.4.

$$R_Q^{c1} := \Pi_{(A_1.*)}(\sigma_{A_1.\text{name} = \text{"foo"}}(\rho_{A_1}(\text{Method})))$$

$$R_Q^{c2} := \Pi_{(A_1.*)}(\sigma_{\Theta_2}(\rho_{A_1}(\text{Method}) \times \rho_{A_2}(\text{Type}) \times \rho_{A_3}(\text{Parameter}) \times \rho_{A_4}(\text{Type})))$$

Here, $\Theta_2 := \Theta \wedge (A_1.\text{name} = \text{"foo"})$ and Θ is shown in Example 3.2. Given the function h shown in Example 5.7, we can obtain that $\alpha_h(R_Q^{c1}, s) = \frac{1}{4}$, $\alpha_h(R_Q^{c2}, s) = 1$, $\beta(R_Q^{c1}) = 2$, and $\beta(R_Q^{c2}) = 10$. According to Example 5.7, we have $R_Q^{c1} \preceq_s R_Q^{c2} \preceq_s R_Q$.

■ **Algorithm 3** Blending selection with refinement.

```

1 Procedure synthesize( $\Gamma, \mathcal{R}, R_p^*, R_n^*, s, K$ ):
2    $\mathcal{R}' \leftarrow \text{reduce}(\Gamma, \mathcal{R}, R_p^*, R_n^*)$ ;
3    $\tilde{\alpha} \leftarrow \frac{1}{|N(s)|} |\{w \in h(R, a) \cap N(s) \mid \exists R \in \mathcal{R}', T \in \mathcal{R}' \cup \{\text{STR}\} : (a, T) \in \Gamma(R)\}|$ ;
4    $\alpha_{max} \leftarrow \text{MIN\_INT}$ ;  $\beta_{min} \leftarrow \text{MAX\_INT}$ ;  $\mathcal{S}_Q \leftarrow \emptyset$ ;
5   foreach  $1 \leq m \leq K \cdot |\mathcal{R}'|$  :
6     if  $\mathcal{S}_R(m-1, k-1) = \emptyset$  and  $\mathcal{S}_R(m-1, k) = \emptyset$  :
7       continue ;
8     foreach  $1 \leq k \leq \min(K, m)$  :
9       refine( $\mathcal{S}_R, \mathcal{S}_C, R_p^*, R_n^*, m, k, \mathcal{R}'$ );
10      foreach  $R_Q \in \mathcal{S}_C(m, k)$  :
11         $(\alpha_{max}, \beta_{min}, \mathcal{S}_Q) \leftarrow \text{update}(R_Q, \alpha_{max}, \beta_{min}, \mathcal{S}_Q)$ ;
12         $\tilde{\beta} = \min(\{\beta(R_Q) \mid R_Q \in \mathcal{S}_R(m, k) \cup \mathcal{S}_R(m, k-1)\})$ ;
13        if  $\alpha_{max} = \tilde{\alpha}$  and  $\beta_{min} \leq \tilde{\beta}$  :
14          return  $\mathcal{S}_Q$ ;
15  return  $\mathcal{S}_Q$ ;
```

The total order is an adaption of Occam’s razor for our synthesis problem. Without the named entity coverage, we would select the query candidates with the lowest structural complexity, such as R_Q^{c1} in Example 5.8, even if they do not constrain the relationship of several grammatical constructs as expected. Based on the total order, we can select the query candidates by solving the dual-objective optimization problem, which finally yields the desired queries for code search.

5.3.2 Blending Selection with Refinement

Based on Definition 5.8, we propose Algorithm 3 that blends the candidate selection with the bounded refinement, which is more likely to obtain the desired queries for a code search task. First, we obtain the schema graph and remove the dummy relations via the representation reduction (line 2). We then compute the upper bound of the named entity coverage, which is denoted by $\tilde{\alpha}$ (line 3). After the initialization of α_{max} , β_{min} , and \mathcal{S}_Q (line 4), we conduct the bounded refinement and select the query candidates in each round (lines 5–14). Obviously, there are at most $K \cdot |\mathcal{R}'|$ relations in a query for a given multiplicity bound K (line 5), and a relation can only appear at most $\min(K, m)$ times in a query with m relations (line 8). In each round, we fuse the refinement and selection as follows:

- Enumerate (m, k) -bounded refinable queries and query candidates with Algorithm 2, strengthening the selection conditions of the refinable queries in previous rounds (line 9).
- Compute $\alpha_h(R_Q, s)$ and $\beta(R_Q)$ for each (m, k) -bounded query candidate R_Q and update the selected candidate set \mathcal{S}_Q , α_{max} , and β_{min} (lines 10–11). Particularly, α_{max} and β_{min} are updated to identify the largest candidates with respect to the total order.
- Terminate the iteration in advance and return the set \mathcal{S}_Q if α_{max} reaches the upper bound of the named entity coverage, i.e., $\tilde{\alpha}$, and the queries to be refined in the next round do not have lower structural complexities than β_{min} (lines 13–14).

The refinement strengthens the selection conditions to exclude all the negative tuples. Specifically, we explore the bounded search space containing the query graphs of refinable queries, avoiding the unnecessary enumerative search effectively. In real-world code search

tasks, the selection condition is often involved different kinds of grammatical constructs, making each relation appear often appear in the conjunction query one or two times. Therefore, we set the multiplicity bound K to 2 for real-world code search tasks in practice, of which the effectiveness will be evidenced by our evaluation.

The natural language description benefits our synthesis process from two aspects. First, the selected queries in \mathcal{S}_Q are the largest queries under the total order, and thus they are more likely to conform to the user’s search intent than other query candidates. Second, we terminate the enumerative search if the named entity coverage cannot increase with a smaller structural complexity, avoiding unnecessary enumerative search of bounded query candidates for the efficiency improvement.

► **Example 5.9.** Consider R_Q^{c1} , R_Q^{c2} and R_Q in Example 5.8. We obtain the query candidate R_Q^{c1} when $(m, k) = (1, 1)$, and discover the candidates R_Q and R_Q^{c2} when $(m, k) = (4, 2)$. Based on Definition 5.8, we select and maintain the query candidate R_Q in \mathcal{S}_Q . Also, we find $\alpha_{max} = \alpha_h(R_Q, s) = 1$ reaches $\tilde{\alpha}$, indicating that the candidates in the subsequent rounds cannot yield a larger named entity coverage with lower structural complexity. Algorithm 3 terminates and returns the set $\mathcal{S}_Q = \{R_Q\}$.

5.4 Summary

Our synthesis algorithm SQUID is an instantiation of a new synthesis paradigm of the multi-modal synthesis, which reduces the synthesis problem to a multi-target optimization problem. We now formulate and prove the soundness, completeness, and optimality of SQUID with three theorems as follows, which are proved in [49].

► **Theorem 5.2 (Soundness).** For any $R_Q \in \mathcal{S}_Q$, where \mathcal{S}_Q is returned by Algorithm 3, R_Q must contain all the positive tuples in R_p^* and exclude the negative tuples in R_n^* .

► **Theorem 5.3 (Completeness).** If an MMCQS problem instance has an (m, k) -bounded query as its solution and $k \leq K$, the set \mathcal{S}_Q returned by Algorithm 3 is not empty.

► **Theorem 5.4 (Optimality).** Denote $I = \{(m, k) \mid 1 \leq m \leq K \cdot |\mathcal{R}'|, 1 \leq k \leq \min(K, m)\}$ and $\tilde{\mathcal{S}} = \bigcup_{(m,k) \in I} \mathcal{S}_C(m, k)$. The returned query set \mathcal{S}_Q of Algorithm 3 satisfies:

- $R'_Q \preceq_s R_Q$ for every $R_Q \in \mathcal{S}_Q$ and $R'_Q \in \tilde{\mathcal{S}}$.
- There do not exist $R_Q \in \tilde{\mathcal{S}} \setminus \mathcal{S}_Q$ and $R'_Q \in \tilde{\mathcal{S}}$ such that $R'_Q \preceq_s R_Q$ and $R_Q \not\preceq_s R'_Q$.

6 Implementation

Established upon the industrial Datalog-based Java program analyzer in Ant Group, SQUID synthesizes conjunctive queries to support code search tasks in Java programs. Noting that our approach is general enough to support the conjunctive query synthesis for any Datalog-based analyzer as long as the generated relations can be formulated by Definition 3.2. In what follows, we provide more implementation details of SQUID.

Synthesis Input Configuration. We design a user interface to convenience the users to specify examples in a code snippet. Specifically, the users can copy a desired grammatical construct from their workspace as a positive example or write a positive example manually. By mutating a positive example, the users can create more positive and negative examples, eventually forming an example program. Then we convert the program to the relational representation, which consists of 173 relations with 1,093 attributes in total, and partitions a

relation into two parts to induce positive and negative tuples. To extract the named entities from the natural language description, we leverage the named entity recognition [31] and construct the dictionary of entities to filter unnecessary named entities in the post-processing. Specifically, the dictionary contains 205 words, which are the keywords describing grammatical constructs in Java programs, such as “method”, “parameter”, and “return”. Furthermore, we also instantiate the function h in Definition 5.6 to bridge the program relational representation with natural language words. We publish all the synthesis specifications and dictionary of entities online [48].

Synthesis Algorithm Design. Based on the language schema of Java, we construct the schema graph offline and persist it for synthesizing queries for a given synthesis specification. Instead of invoking the Datalog-based analyzer, we implement a query evaluator for conjunctive queries upon the relational representation to identify the refinable queries and query candidates, which can improve the efficiency of the query evaluation during the synthesis. In the bounded refinement, we set the multiplicity bound K to 2 by default to support code search tasks. To efficiently synthesize string constraints, we leverage the generalized suffix automaton [33] to identify the longest common substrings of a set of string values, which returns the string predicate p and the string literal ℓ with low time overhead. Currently, SQUID utilizes four predicates for string match, while we can further extend it to support regex match by adopting existing regex synthesis techniques [27, 10] to Algorithm 2.

7 Evaluation

To quantify the effectiveness and efficiency of SQUID, we conduct a comprehensive empirical evaluation and answer the following four research questions:

- **RQ1:** How effective is SQUID in the conjunctive query synthesis for code search tasks?
- **RQ2:** How big are the benefits of the representation reduction and the bounded refinement in terms of efficiency?
- **RQ3:** Is the query candidate selection effective and necessary for the synthesis?
- **RQ4:** How does SQUID compare to other approaches that could be used in our problem?

Benchmark. There are no existing studies targeting our multi-modal synthesis problem, so we construct a new benchmark for evaluation, which consists of 31 code search tasks. As shown in Table 1, the tasks cover five kinds of grammatical constructs, namely variables, expressions, statements, methods, and classes. Specifically, 14 tasks are the variants of C++ search tasks in [34] or the query synthesis tasks in [46]. We also consider more advanced tasks deriving from real demands. For example, Task 2 originates from the coding standard of a technical unit in Ant Group, while Task 21 is often conducted when the developers check the usage of the `log4j` library to improve reliability. For each task, we specify examples in a program and a sentence as the natural language description. The program is fed to the commercial Datalog-based analyzer in Ant Group to generate the relational representation and the relation partition, while the natural language description is processed via the named entity recognition technique [31]. The columns **L** and **(P, N)** in Table 1 indicate the line numbers of the programs and the numbers of positive/negative tuples, respectively.

Experimental Setup. We conduct all the experiments on a Macbook Pro with a 2.6 GHz Intel® Core™ i7-9750H CPU and 16 GB physical memory.

■ **Table 1** Experiment results of synthesizing conjunctive queries for code search tasks.

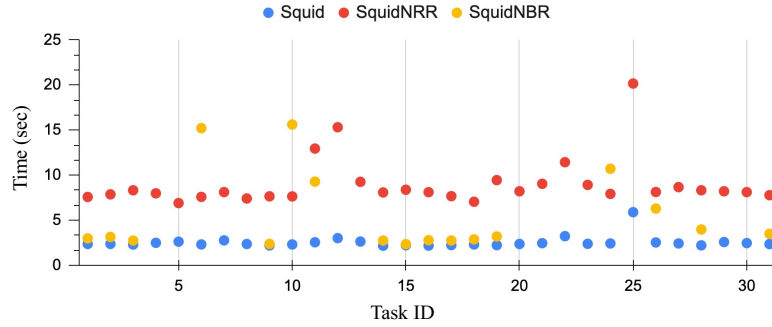
ID	Description	L	(P, N)	$ G_Q $	k	$ G'_T $	T_0 (s)
1	Local variables with double type	10	(2, 2)	(2, 1, 1)	1	(10, 19)	2.36
2	Float variables of which the identifier contains “cash”	10	(3, 1)	(3, 2, 2)	1	(9, 17)	2.37
3	Public field variables of a class	6	(2, 1)	(2, 1, 1)	1	(10, 21)	2.30
4	Public field variables whose names use “cash” as suffixes	8	(3, 2)	(2, 2, 2)	1	(10, 23)	2.49
5	Arithmetic expressions using double-type operands	9	(2, 2)	(3, 2, 2)	1	(9, 29)	2.62
6	Cast expressions from double-type to float-type [46]	11	(1, 2)	(3, 2, 2)	2	(9, 23)	2.31
7	Arithmetic expressions only using literals as operands	19	(2, 3)	(3, 2, 1)	2	(9, 29)	2.76
8	Expressions comparing a variable and a boolean literal [34]	16	(2, 1)	(2, 1, 1)	1	(8, 29)	2.36
9	New expressions of ArrayList	8	(2, 1)	(2, 1, 1)	1	(10, 25)	2.20
10	Logical conjunctions with a boolean literal [34]	11	(3, 1)	(2, 1, 2)	1	(9, 29)	2.31
11	Float increment expression [46]	11	(1, 2)	(3, 2, 2)	1	(9, 23)	2.55
12	Expressions comparing two strings with “==” [34]	14	(2, 1)	(3, 2, 3)	1	(11, 46)	3.01
13	Expressions performing downcasting [46]	25	(2, 1)	(3, 2, 0)	1	(11, 39)	2.63
14	The import of LocalTime	7	(1, 1)	(1, 0, 2)	1	(9, 23)	2.17
15	The import of the classes in log4j	9	(3, 1)	(1, 0, 1)	1	(9, 22)	2.22
16	Labeled statements using “err” as the label [34]	17	(1, 1)	(1, 0, 1)	1	(10, 21)	2.18
17	If-statements with a boolean literal as a condition [34]	16	(2, 1)	(2, 1, 0)	1	(9, 17)	2.24
18	For-statements with a boolean literal as the condition [34]	15	(2, 1)	(2, 1, 0)	1	(10, 25)	2.31
19	Invocation of unsafe time function “localtime” [34]	9	(2, 1)	(2, 1, 1)	1	(10, 23)	2.23
20	Public methods with void return type [34]	10	(2, 1)	(3, 2, 2)	1	(11, 26)	2.36
21	Methods receiving a parameter with Log4jUtils type	11	(2, 1)	(3, 2, 1)	1	(9, 20)	2.45
22	Methods using a boolean parameter as a if-condition [46]	29	(2, 2)	(4, 3, 0)	1	(11, 26)	3.23
23	Methods creating a File object	14	(2, 1)	(3, 2, 1)	1	(12, 30)	2.38
24	Mutually recursive methods [34, 46]	20	(2, 2)	(2, 2, 0)	2	(11, 27)	2.42
25	Overriding methods of classes [46]	25	(2, 4)	(5, 5, 0)	2	(8, 22)	5.89
26	User classes with “login” methods	15	(2, 1)	(2, 1, 2)	1	(11, 28)	2.53
27	Classes containing a field with Log4jUtils type	20	(2, 1)	(3, 2, 1)	1	(12, 35)	2.42
28	Classes having a subclass	25	(3, 3)	(2, 1, 0)	2	(13, 33)	2.21
29	Classes implementing Comparable interface	16	(2, 1)	(2, 1, 1)	1	(13, 35)	2.58
30	Classes containing a static method	17	(2, 1)	(3, 2, 1)	1	(11, 28)	2.46
31	Java classes with main functions	16	(2, 1)	(2, 1, 1)	1	(10, 28)	2.35

7.1 Overall Effectiveness

To evaluate the effectiveness of SQUID, we run it upon the synthesis specification for each code search task, examining whether the synthesized queries express the intent correctly, and meanwhile, measure the time cost of synthesizing queries in each task.

In Table 1, the column $|G_Q|$ indicates the numbers of the relations, equality constraints, and string constraints. The column k shows the maximal multiplicity of a relation in a synthesized query, while the column $|G'_T|$ indicates the numbers of nodes and edges in the subgraph of the schema graph induced by $\mathcal{R}' \cup \{\text{STR}\}$. The time cost of SQUID is shown in the column T_0 . According to the statistics, we can obtain two main findings. First, SQUID synthesizes the queries for all the tasks successfully. It manipulates more than three relations in Tasks 22 and 25, which are even non-trivial for a human to achieve. Second, SQUID synthesizes the queries with a quite low time cost. The average time cost is 2.56 seconds, while most of the tasks are finished in three seconds.

As mentioned in § 6, SQUID performs the bounded refinement with the multiplicity bound $K = 2$. In our benchmark, five code search tasks demand several relations appear two times. In practice, the searching condition can hardly relate to more than two grammatical constructs of the same kind, so our setting of the multiplicity bound K enables SQUID to synthesize



■ **Figure 5** The time cost comparison of SQUID, SQUIDNRR, and SQUIDNBR.

queries for code search tasks in real-world scenarios. Meanwhile, we quantify the time cost of the synthesis in the cases of $K = 3$ and $K = 4$. Averagely, SQUID takes 2.71 seconds and 3.98 seconds under the two settings, respectively. Thus, the overhead increases gracefully when K increases, demonstrating the great potential of SQUID in efficiently synthesizing more sophisticated queries with a larger multiplicity bound.

7.2 Ablation Study on Efficiency

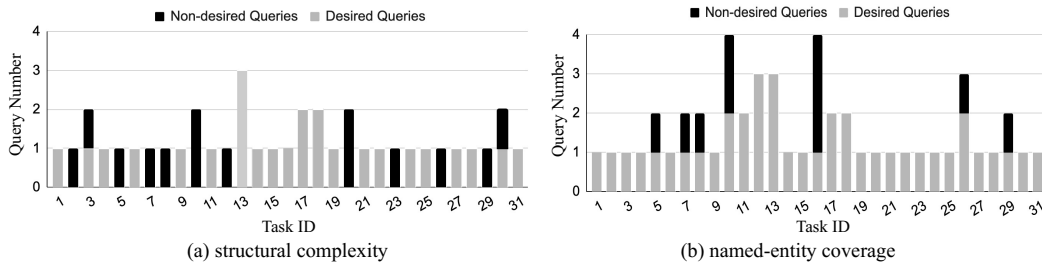
We evaluate two ablations of SQUID, namely SQUIDNRR and SQUIDNBR, to quantify the impact of the representation reduction and the bounded refinement on the efficiency.

- SQUIDNRR: This ablation of SQUID does not perform the representation reduction but still leverages Algorithm 2 to conduct the bounded refinement.
- SQUIDNBR: The ablation performs the representation reduction as SQUID does, while it enumerates all the query graphs and permits each relation to appear at most K times.

We measure the time cost of two ablations to quantify their efficiency. Specifically, we set the time budget for synthesizing queries for a single task to 30 seconds, as a synthesizer would have little practical value for the real-world code search if it ran out of the time budget.

Figure 5 shows the comparison of the time cost of SQUID and the two ablations. First, the representation reduction can effectively reduce the time cost. Specifically, the average time cost of SQUIDNRR is 8.98 seconds, indicating that the representation reduction introduces a 71.49% reduction over the time cost. Second, the bounded refinement has a critical impact on the efficiency of SQUID. Without the refinement, SQUIDNBR has to explore the huge search space induced by the non-dummy relations, making 14 out of 31 tasks cannot be finished within the time budget, such as Task 4, Task 5, etc. For the failed tasks, we do not show the time cost of SQUIDNBR in Figure 5. SQUIDNBR also takes much more time than SQUID, consuming 7.89 seconds on average, even if it successfully synthesizes the queries.

To investigate how the efficiency is improved, we further measure the size of the subgraph of the schema graph induced by $\mathcal{R}' \cup \{\text{STR}\}$. Initially, the schema graph contains 174 nodes (including the node depicting STR) and 1,093 edges. As shown in the column $|\mathbf{G}'_{\mathbf{T}}|$ of Table 1, the induced subgraph only contains around ten nodes and no more than fifty edges. Although SQUIDNRR prunes unnecessary relations by enumerating several bounded queries at the beginning of the refinement, it has to spend more time on the query enumeration than SQUID, which demonstrates the critical role of the bounded refinement in our synthesis. Besides, the running time of SQUIDNBR is similar to SQUID on several benchmarks, such as Task 1, Task 2, Task 3, Task 9, etc., while it takes much longer time than SQUID in other benchmarks.



■ **Figure 6** The numbers of synthesized queries prioritized with different metrics.

Although SQUIDNBR does not discard infeasible queries, it still benefits from representation reduction. When a desired query is of small size and the reduced program representation induces a small schema graph G'_T , SQUIDNBR can terminate to find an optimal query by enumerating a few candidates. However, if G_Q and G'_T are large, SQUIDNBR enumerates a large number of infeasible queries, which introduces significant overhead.

7.3 Impact of Selection

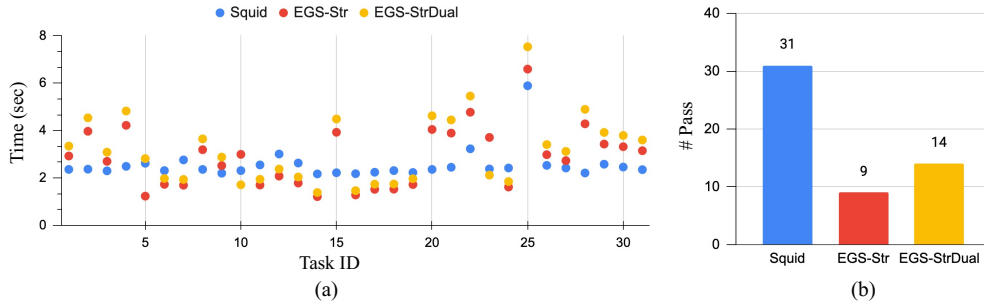
To measure the impact of the query candidate selection, we adapt each metric separately for candidate prioritization. Specifically, we alter Algorithm 3 and select the queries minimizing the structural complexity and maximizing the named entity coverage, respectively. We then count the returned queries and inspect whether they are desired ones or not.

Figure 6(a) shows the numbers of the synthesized queries with the structural complexity as the metric. As we can see, SQUID produces non-desired queries in 12 tasks, while the returned set of synthesized queries in 10 tasks do not contain any desired query. For Task 3 and Task 30, it provides the desired queries along with non-desired ones, which makes the users confused about how to select a proper one. Similar to R_Q^{c1} in Example 5.8, the non-desired queries are caused by the over-fitting of positive and negative examples. Although they have the simplest form of the selection conditions, the relationship of grammatical constructs mentioned in the natural language description is not constrained, making synthesized queries not express the users' intent correctly.

Figure 6(b) shows the numbers of the queries maximizing the named entity coverage. SQUID returns at least one non-desired query for seven tasks. Similar to R_Q^{c2} in Example 5.8, non-desired queries come from over-complicated selection conditions. Although the selected queries have the same named entity coverage, several queries contain more atomic formulas than the desired ones, posing stronger restrictions upon the code. Besides, the synthesized queries in several tasks, e.g., Task 11 and Task 26, have complex selection conditions although they are equivalent under the context of code search. However, such queries exhibit higher structural complexity, posing more difficulty in understanding them.

7.4 Comparison with Existing Techniques

To the best of our knowledge, no existing technique or implemented tool targets the same problem as SQUID. To compare SQUID with existing effort, we adapt the state-of-the-art Datalog synthesizer EGS [46] as our baseline. Originally, it synthesizes a conjunctive query to separate a positive tuple from all the negative ones and then group all the conjunctive queries as the final synthesis result, which can be theoretically a disjunctive query. However,



■ **Figure 7** The time cost and the numbers of passed tasks of SQUID, EGS-STR, and EGS-STRDUAL.

EGS does not synthesize string constraints and only prioritizes feasible solutions based on their sizes, i.e., the structural complexity in our work. Thus, we construct two adaptations, namely EGS-STR and EGS-STRDUAL, to synthesize the queries under our problem setting.

- EGS-STR computes the longest substring of each string attribute in the positive tuple such that the string values of the attributes in negative ones do not contain it as the substring. We follow the priority function in EGS, which consists of the number of undesirable tuples eliminated per atomic constraint and the size of a query, to accelerate searching a query candidate with a small size. Finally, it obtains a query candidate for each positive tuple and groups the candidates as a result.
- EGS-STRDUAL further extends EGS-STR by considering the named entity coverage. Specifically, it prioritizes the refinable queries according to the three metrics, including the number of undesirable tuples eliminated per atomic constraint, the named entity coverage, and the size of a query. Other settings are the same as the ones of EGS-STR.

Figure 7 shows the results of the comparison. On average, EGS-STR and EGS-STRDUAL spend 2.85 and 3.18 seconds on a synthesis instance, respectively, while the average time cost of SQUID is 2.56 seconds. Although EGS-STR and EGS-STRDUAL accelerate the searching process with priority functions as the heuristic metrics, they have to process the positive tuples in each round, and thus, the number of positive tuples can increase the overhead.

Meanwhile, the two baselines only succeed synthesizing queries for 9 and 14 tasks, respectively. There are two root causes of their failures in synthesizing desired queries. First, they synthesize the query candidates for each positive query separately and, thus, are more prone to over-fitting problems than SQUID. Second, the core algorithms of EGS and the two adaptations, which are the instantiations of inductive logic programming, can not guarantee the obtained solutions are optimal under the given metrics. As reported in [46], the query may not be of the minimal size if EGS leverages the number of undesirable tuples eliminated per atomic constraint to accelerate the searching process of a query candidate. In our problem, our dual quantitative metrics increase the difficulty of achieving the optimal solutions with the two adaptations, which causes the failures of the code search tasks.

7.5 Discussion

In what follows, we demonstrate the discussions on the limitations of SQUID and several future works, which can further improve the practicality of our techniques.

Limitations. Although SQUID is demonstrated to be effective for code search, it has two major limitations. First, SQUID cannot synthesize the query where the multiplicity of a relation is larger than the multiplicity bound K . In other words, Theorem 5.3 actually ensures partial completeness. Although we may achieve completeness for all realizable instances by enumerating queries until obtaining a query candidate in Algorithm 3, SQUID would fail to terminate for unrealizable instances. Second, SQUID does not support synthesizing the queries with logical disjunctions. However, when a code search task involves the matching of multiple patterns, SQUID would not discover the correct queries, which are out of the scope of the conjunctive queries.

Future Works. In the future, we will attempt to propose an efficient decision procedure to identify unrealizable instances. Equipped with the decision procedure, we can only enumerate queries for realizable instances and generalize the query refinement by discarding the multiplicity bound. Besides, it would be promising to generalize SQUID for disjunctive query synthesis. One possible adaptation is to divide positive tuples into proper clusters and synthesize a conjunctive part for each cluster separately, following existing studies such as EGS [46] and RHOSYNTH [15]. In addition, we aim to expand SQUID to diverse program domains, such as serverless functions [50] and programs running on networking devices [57]. These use cases have gained significant attention in recent years, which can pose new challenges on code search where new approaches may be needed. Lastly, it would be meaningful to combine SQUID with the techniques in the community of human-computer interaction [8] to unleash its benefit for practice use.

8 Related Work

Multi-modal Program Synthesis. There has been a vast amount of literature on the multi-modal synthesis [11, 4, 9, 10, 40, 16, 34]. For example, the LTL formula synthesizer LTLTALK [16] maximizes the objective function that measures the similarity between the natural language description and the LTL formula, and searches for the optimal solution that distinguishes the positive and negative examples. SQUID bears similarities to LTLTALK in terms of the prioritization, while we use the named entities to avoid the failure of semantic parsing of a sentence. Another closely related work is a query synthesizer named SPORQ [34]. Based on code examples and user feedback, SPORQ iterates its PBE-based synthesis engine to refine the queries, which demands verbose user interactions and a long time period. In contrast, SQUID automates the code search by solving a new multi-modal synthesis problem, which only requires the users to specify code examples and a natural language description, effectively relieving the user's burden in the searching process.

Component-based Synthesis. Several recent studies aim to compose several components (e.g., the classes and methods in the libraries) into programs that achieve target functionalities [25, 18, 24, 38, 20, 21]. Typically, SyPET [14] and APIPHANY [19] both use the Petri net to encode the type signature of each function, and collect the reachable paths to enumerate the well-typed sketches of the programs, which prunes the search space at the start of the synthesis. In our work, SQUID leverages the schema graph to guide the enumerative search, which share the similarity with existing studies. However, our enumerative search space does not consist of the reachable paths in the schema graph, and instead, contains different choices of selecting its nodes and edges. Besides, unlike prior efforts [14, 24, 20, 19], SQUID computes the activated relations and then discards unnecessary components, i.e., dummy relations, which distinguishes SQUID significantly from other component-based synthesizers.

Datalog Program Synthesis. There have been many existing efforts of synthesizing Datalog programs [2, 42, 39, 43, 32]. For example, ZAATAR [2] encodes the input-output examples and Datalog programs with SMT formulas, and synthesizes the candidate solution via constraint solving. Unlike constraint-based approaches, ALPS [42] and GENSYNTH [32] synthesize target Datalog programs via the enumerative search, which is similar to our synthesis algorithm. However, existing studies do not tackle a large number of relations in the synthesis [2, 42, 32] or pursue an optimal solution with respect to a natural language description. Meanwhile, they do not support the synthesis of string constraints, making their approaches incapable of string matching-based code search. In contrast, SQUID ensures soundness, completeness, and optimality simultaneously and synthesizes string constraints for string matching, showing its potential in assisting real-world code search tasks.

Datalog-based Program Analysis. The past few decades have witnessed the increasing popularity of Datalog-based program analysis [22, 52, 55, 3, 44]. For example, CODEQL encodes a program with a relational representation and exposes a query language for query writing [3]. Several analyzers target more advanced semantic reasoning. For example, the points-to and alias facts are depicted by two kinds of relations in DOOP [6], and meanwhile, pointer analysis algorithms are instantiated as Datalog rules [44]. Other properties, such as def-use relation and type information, can also be analyzed by existing analyzers [26, 37]. Our effort has shown the opportunity of unleashing the power of Datalog-based program analyzers seamlessly to support the code search automatically.

9 Conclusion

We propose an efficient synthesis algorithm SQUID for a multi-modal conjunctive query synthesis problem, which enables automatic code search using a Datalog-based program analyzer. SQUID reduces the search space via the representation reduction and the bounded refinement, and meanwhile, conducts the query candidate selection with dual quantitative metrics. It efficiently synthesizes the queries for 31 code search tasks with the guarantees of soundness, completeness, and optimality. Its theoretical and empirical results offer strong evidence of its practical value in assisting code search in real-world scenarios.

References

- 1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: <http://webdam.inria.fr/Alice/>.
- 2 Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 689–706. Springer, 2017. doi:10.1007/978-3-319-66158-2_44.
- 3 Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPICs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.ECOOP.2016.2.
- 4 Christopher Baik, Zhongjun Jin, Michael J. Cafarella, and H. V. Jagadish. Duoquest: A dual-specification system for expressive SQL queries. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2319–2329. ACM, 2020. doi:10.1145/3318464.3389776.



- 5 Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Inf. Process. Lett.*, 24(6):377–380, 1987. doi:10.1016/0020-0190(87)90114-1.
- 6 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In Shail Arora and Gary T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 243–262. ACM, 2009. doi:10.1145/1640089.1640108.
- 7 Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM, 1977. doi:10.1145/800105.803397.
- 8 Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. PL and HCI: better together. *Commun. ACM*, 64(8):98–106, 2021. doi:10.1145/3469279.
- 9 Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. Web question answering with neurosymbolic program synthesis. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 328–343. ACM, 2021. doi:10.1145/3453483.3454047.
- 10 Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 487–502. ACM, 2020. doi:10.1145/3385412.3385988.
- 11 Yanju Chen, Ruben Martins, and Yu Feng. Maximal multi-layer specification synthesis. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 602–612. ACM, 2019. doi:10.1145/3338906.3338951.
- 12 Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM, 2016. doi:10.1145/2970276.2970347.
- 13 Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436. ACM, 2017. doi:10.1145/3062341.3062351.
- 14 Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612. ACM, 2017. doi:10.1145/3009837.3009851.
- 15 Pranav Garg and Srinivasan H. Sengamedu. Synthesizing code quality rules from examples. *Proc. ACM Program. Lang.*, 6(OOPSLA2), October 2022. doi:10.1145/3563350.
- 16 Ivan Gavran, Eva Darulova, and Rupak Majumdar. Interactive synthesis of temporal specifications from examples and natural language. *Proc. ACM Program. Lang.*, 4(OOPSLA):201:1–201:26, 2020. doi:10.1145/3428269.
- 17 Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006. doi:10.1145/1131342.1131345.
- 18 Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 50–61. ACM, 2011. doi:10.1145/1993498.1993505.

- 19 Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 122–136. ACM, 2022. doi:10.1145/3519939.3523450.
- 20 Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL):12:1–12:28, 2020. doi:10.1145/3371080.
- 21 Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38. ACM, 2013. doi:10.1145/2491956.2462192.
- 22 Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *codeQuest: scalable source code queries with datalog*. In Dave Thomas, editor, *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006. doi:10.1007/11785477_2.
- 23 IntelliJ IDEA. Structural search and replace, <https://www.jetbrains.com/help/idea/structural-search-and-replace.html>, 2022. [Online; accessed 10-Nov-2022]. URL: <https://www.jetbrains.com/help/idea/structural-search-and-replace.html>.
- 24 Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):205:1–205:27, 2020. doi:10.1145/3428273.
- 25 Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM, 2010. doi:10.1145/1806799.1806833.
- 26 Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12, 2005.
- 27 Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In Bernd Fischer and Ina Schaefer, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 70–80. ACM, 2016. doi:10.1145/2993236.2993244.
- 28 Tao Lei, Fan Long, Regina Barzilay, and Martin C. Rinard. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 1294–1303. The Association for Computer Linguistics, 2013. URL: <https://aclanthology.org/P13-1127/>.
- 29 Xuan Li, Zerui Wang, Qianxiang Wang, Shoumeng Yan, Tao Xie, and Hong Mei. Relationship-aware code search for javascript frameworks. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 690–701. ACM, 2016. doi:10.1145/2950290.2950341.
- 30 Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John C. Grundy. Opportunities and challenges in code search tools. *ACM Comput. Surv.*, 54(9):196:1–196:40, 2022. doi:10.1145/3480027.
- 31 Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, System Demonstrations*, pages 55–60. The Association for Computer Linguistics, 2014. doi:10.3115/v1/p14-5010.

- 32 Jonathan Mendelson, Aaditya Naik, Mukund Raghothaman, and Mayur Naik. GENSYNTH: synthesizing datalog programs without language bias. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pages 6444–6453. AAAI Press, 2021. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/16799>.
- 33 Mehryar Mohri, Pedro J. Moreno, and Eugene Weinstein. General suffix automaton construction algorithm and space bounds. *Theor. Comput. Sci.*, 410(37):3553–3562, 2009. doi:10.1016/j.tcs.2009.03.034.
- 34 Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. Sporq: An interactive environment for exploring code using query-by-example. In Jeffrey Nichols, Ranjitha Kumar, and Michael Nebeling, editors, *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, pages 84–99. ACM, 2021. doi:10.1145/3472749.3474737.
- 35 Mayur Naik. Chord: A versatile platform for program analysis. In *Tutorial at ACM Conference on Programming Language Design and Implementation*, 2011.
- 36 Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D’Antoni. Automatic repair of regular expressions. *Proc. ACM Program. Lang.*, 3(OOPSLA):139:1–139:29, 2019. doi:10.1145/3360565.
- 37 Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. Codetrek: Flexible modeling of code using an extensible relational representation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: <https://openreview.net/forum?id=WQc075jmBmf>.
- 38 Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286. ACM, 2012. doi:10.1145/2254064.2254098.
- 39 Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, 4(POPL):62:1–62:27, 2020. doi:10.1145/3371130.
- 40 Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 792–800. AAAI Press, 2015. URL: <http://ijcai.org/Abstract/15/117>.
- 41 Logging Services. Apache log4j security vulnerabilities , <https://logging.apache.org/log4j/2.x/security.html>, 2021. [Online; accessed 10-Nov-2022]. URL: <https://logging.apache.org/log4j/2.x/security.html>.
- 42 Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of datalog programs. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 515–527. ACM, 2018. doi:10.1145/3236024.3236034.
- 43 Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing datalog programs using numerical relaxation. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6117–6124. ijcai.org, 2019. doi:10.24963/ijcai.2019/847.
- 44 Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 245–251. Springer, 2010. doi:10.1007/978-3-642-24206-9_14.

- 45 CODEQL. CodeQL for Java. <https://codeql.github.com/docs/codeql-language-guides/codeql-for-java/>, 2022. [Online; accessed 10-Nov-2022].
- 46 Aalok Thakkar, Aaditya Naik, Nathaniel Sands, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. Example-guided synthesis of relational queries. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1110–1125. ACM, 2021. doi:10.1145/3453483.3454098.
- 47 Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in C. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 752–762. ACM, 2017. doi:10.1145/3106237.3106300.
- 48 SQUID. SquidData. <https://github.com/SquidData/SquidData>, 2022. [Online; accessed 10-Nov-2022].
- 49 Chengpeng Wang, Peisen Yao, Wensheng Tang, Gang Fan, and Charles Zhang. Synthesizing conjunctive queries for code search. *CoRR*, abs/2305.04316, 2023. doi:arXiv.2305.04316.
- 50 Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable nf virtualization platform. In *Proceedings of the 13th Symposium on Cloud Computing, SoCC '22*, pages 493–509, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3542929.3563471.
- 51 Brendon J Wilson. Java coding convention, 2000.
- 52 Xiuheng Wu, Chenguang Zhu, and Yi Li. DIFFBASE: a differential factbase for effective software evolution management. In Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 503–515. ACM, 2021. doi:10.1145/3468264.3468605.
- 53 Yingfei Xiong and Bo Wang. L2S: A framework for synthesizing the most probable program under a specification. *ACM Trans. Softw. Eng. Methodol.*, 31(3):34:1–34:45, 2022. doi:10.1145/3487570.
- 54 Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA):63:1–63:26, 2017. doi:10.1145/3133887.
- 55 Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 590–604. IEEE Computer Society, 2014. doi:10.1109/SP.2014.44.
- 56 Junwen Yang, Pranav Subramaniam, Shan Lu, Cong Yan, and Alvin Cheung. How *not* to structure your database-backed web applications: a study of performance bugs in the wild. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 800–810. ACM, 2018. doi:10.1145/3180155.3180194.
- 57 Jane Yen, Jianfeng Wang, Sucha Supittayapornpong, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Meeting slos in cross-platform nfv. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '20*, pages 509–523, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3386367.3431292.
- 58 Xiangyu Zhou, Rastislav Bodík, Alvin Cheung, and Chenglong Wang. Synthesizing analytical SQL queries from computation demonstration. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 168–182. ACM, 2022. doi:10.1145/3519939.3523712.

Do Machine Learning Models Produce TypeScript Types That Type Check?

Ming-Ho Yee  

Northeastern University, Boston, MA, USA

Arjun Guha  

Northeastern University, Boston, MA, USA

Roblox Research, San Mateo, CA, USA

Abstract

Type migration is the process of adding types to untyped code to gain assurance at compile time. TypeScript and other gradual type systems facilitate type migration by allowing programmers to start with imprecise types and gradually strengthen them. However, adding types is a manual effort and several migrations on large, industry codebases have been reported to have taken several years. In the research community, there has been significant interest in using machine learning to automate TypeScript type migration. Existing machine learning models report a high degree of accuracy in predicting individual TypeScript type annotations. However, in this paper we argue that accuracy can be misleading, and we should address a different question: can an automatic type migration tool produce code that passes the TypeScript type checker?

We present TYPEWEAVER, a TypeScript type migration tool that can be used with an arbitrary type prediction model. We evaluate TYPEWEAVER with three models from the literature: DeepTyper, a recurrent neural network; LambdaNet, a graph neural network; and InCoder, a general-purpose, multi-language transformer that supports fill-in-the-middle tasks. Our tool automates several steps that are necessary for using a type prediction model, including (1) importing types for a project's dependencies; (2) migrating JavaScript modules to TypeScript notation; (3) inserting predicted type annotations into the program to produce TypeScript when needed; and (4) rejecting non-type predictions when needed.

We evaluate TYPEWEAVER on a dataset of 513 JavaScript packages, including packages that have never been typed before. With the best type prediction model, we find that only 21% of packages type check, but more encouragingly, 69% of files type check successfully.

2012 ACM Subject Classification Software and its engineering → Source code generation; General and reference → Evaluation; Theory of computation → Type structures

Keywords and phrases Type migration, deep learning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.37

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.5>

Software (Artifact Evaluation approved artifact): <https://doi.org/10.5281/zenodo.7662708>

Software (code repository): <https://github.com/nuprl/TypeWeaver>

archived at [sw:h1:dir:34399ede560aa59cfe736bf9994185d54b8c2e7e](https://sw.h1.dir:34399ede560aa59cfe736bf9994185d54b8c2e7e)

Funding This work is partially supported by the National Science Foundation grant CCF-2102291.

Acknowledgements We thank Northeastern Research Computing and the New England Research Cloud for providing computing resources; and Leif Andersen, Luna Phipps-Costin, Donald Pinckney, and the anonymous reviewers for their feedback.



© Ming-Ho Yee and Arjun Guha;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 37; pp. 37:1–37:28



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Gradual typing allows programmers to freely mix statically and dynamically typed code. This makes it possible to add static types to a large program incrementally, and slowly reap the benefits of static typing without requiring a complete rewrite of an existing codebase at once [22, 46, 48]. Over the past decade, gradual typing has proliferated, and there are now gradually typed versions of several mainstream languages [6, 7, 11, 14, 31, 33, 37, 48, 52].

TypeScript is a widely used gradually typed language, and a syntactic superset of JavaScript. Programmers can write their code in TypeScript, benefit from static typing, and then compile to JavaScript. However, the process of *migrating* an untyped JavaScript program to TypeScript has remained a labor-intensive manual effort in practice. For example, Airbnb engineers took more than two years to add TypeScript type annotations to 6 million lines of JavaScript [44], and there are several other accounts of multi-year TypeScript migration efforts [3, 8, 36, 39, 43].

To address this problem, there has been significant research interest in using machine learning to predict TypeScript types. Machine learning seems attractive because TypeScript has language features (e.g., `eval`) that are very difficult to accommodate with traditional, constraint-based approaches. Moreover, there is a significant quantity of open-source TypeScript that is available to serve as training data for a machine learning model. Over the last few years, advances in model architectures and high-quality training data have led to type annotation prediction with high accuracy on individual type annotations [23, 24, 25, 38, 50].

However, in this paper we argue that accuracy can be misleading, and that predicting individual type annotations is just the first step of migrating a codebase from JavaScript to TypeScript. We should address a different question: *can an automatic type migration tool produce code that type checks?* If so, we prefer type annotations that are non-trivial and useful (i.e., annotations that are not just `any`). On the other hand, if the code does not type check, it may have too many errors, which can overwhelm a user who may just turn off the tool. Moreover, it may not be feasible to fix the type errors automatically, since type errors refer to code locations whose typed terms are used, and not necessarily to faulty annotations.

To answer the type checking question, we present TYPEWEAVER, a TypeScript type migration tool that can be used with an arbitrary type prediction model. Our evaluation employs three models from the literature: DeepTyper [23], a recurrent neural network; LambdaNet [50], a graph neural network; and InCoder [18], a general-purpose, multi-language transformer that supports fill-in-the-middle tasks. Our tool automates several steps that are necessary for using a type prediction model, including:

Importing type dependencies Before migrating a JavaScript project, we must ensure that its dependencies are typed. This means transitively migrating dependencies, or ensuring that the dependencies have TypeScript interface declaration (`.d.ts`) files available.

Module conversion JavaScript code written for Node.js may use either the CommonJS or ECMAScript module system. However, when migrated to TypeScript, only ECMAScript modules preserve type information. Thus, to fully benefit from static type checking, code written with CommonJS modules should be refactored to use ECMAScript modules.

Type weaving Models that assign type labels to variables do not update the JavaScript source to include type annotations. Therefore, to ask if a program type checks, we must “weave” the predicted type annotations with the original JavaScript source to produce TypeScript.

Rejecting non-type predictions Models that predict type annotations as in-filled sequences of tokens can easily produce token sequences that are not syntactic types. These predictions need to be rejected or cleaned for type prediction to work.

```

1 function f(x) { return x+x; }
2 f(1) // returns 2
3 f("a") // returns "aa"
4 var point = {};
5 point.x = 42;
6 point.y = 54;

```

(a) JavaScript function that adds or concatenates its argument to itself.

(b) JavaScript that creates a “point” object.

■ **Figure 1** JavaScript code that cannot be easily typed in TypeScript.

After completing these tasks, it is then possible to type check the resulting TypeScript program and evaluate the effectiveness of TYPEWEAVER.

Our contributions are the following:

- We describe TYPEWEAVER, a TypeScript type migration tool for evaluating type prediction models, which automates several prerequisite steps (Section 3).
- We provide a dataset of 513 JavaScript packages, which are a subset of the top 1,000 most downloaded packages from the npm Registry. Our dataset includes packages without known type annotations, i.e., code that has never been used before to evaluate type prediction models (Section 4.1).
- We report the success rate of type checking. We answer the questions of how many packages type check, how many files type check, how many type annotations are trivial, and whether the predicted types match human-written types (Section 4.2).
- We discuss the common kinds of errors that result from a type migration (Section 4.3).
- We compare the results of a type migration before and after converting to the ECMAScript module system (Section 4.4).
- Finally, we examine four packages as case studies, to showcase other difficulties that arise during type migration (Section 4.5).

2 Background

In this section, we first provide background on the type migration problem and contrast it to type inference. We then discuss deep-learning-based type annotation prediction, focusing on the tools that we have used with TYPEWEAVER.

2.1 Type Migration vs. Type Inference

Type inference is related to, but distinct from, type migration. The goal of type inference is to *reconstruct* the types of variables, expressions, and functions, where some or all the type annotations are missing. In other words, the language is statically typed and the types exist implicitly within the program, so the type inference algorithm computes the missing annotations. Furthermore, inference can frequently compute the *principal type* of a variable, expression, or function, i.e., the most general type. As a result, there is a single answer for the type of a variable, expression, or function. Additionally, in languages that support type inference, the inferred type annotations are well defined and not added to the program text.

In this paper, we use *type migration* to describe the problem of migrating a program from an untyped language to a typed language, e.g., from JavaScript to TypeScript, a process that may require refactoring in addition to type inference. The type migration process starts from an untyped program without type annotations: there is no type information available, beyond the basic information available from literal values, operators, and control-flow statements, so type definitions may need to be inserted into the program. Furthermore, multiple type

annotations may be valid, rather than having a single principal type. For example, Figure 1a shows a JavaScript function where the parameter `x` on Line 1 could be annotated as `number` or `string`; without additional context, both annotations are valid. The example illustrates another challenge of type migration: `f` is called on Line 2 with a number and Line 3 with a string, so the only valid type annotation for `x` is `any`.¹ This satisfies the TypeScript compiler’s type checker, but may not be a helpful annotation in terms of documentation.

The type migration problem for TypeScript has several additional challenges and we highlight some of them here. TypeScript has a structural type system, which makes it even harder to determine the right annotation. Furthermore, structural types are often verbose, making it difficult for a programmer to understand the code, which defeats one of the benefits of a static type system. Another difficulty is that JavaScript code can be too dynamic to fit within TypeScript’s type system, e.g., there is no good way to handle `eval`, other than using the `any` annotation as an escape hatch. Finally, certain idioms and patterns in JavaScript code do not fit TypeScript and need to be refactored. For instance, consider Figure 1b, which initializes a “point” object in JavaScript. Line 4 initializes `point` to an empty object, and Lines 5 and 6 set the `x` and `y` properties. However, this cannot be easily typed in TypeScript,² and it is more appropriate to rewrite the code to use TypeScript classes.

In this landscape of challenges, recent work has focused on the narrower problem of assigning type annotations to TypeScript code, in particular, using deep learning approaches. We examine some of these approaches in the next subsection.

2.2 Deep-Learning-Based Type Annotation Prediction

We focus on JavaScript and TypeScript, since there have been a variety of proposed type prediction models for those languages. We evaluate three of them here: DeepTyper [23], LambdaNet [50], and InCoder [18].

DeepTyper was the first deep neural network for TypeScript type prediction, and uses a *bidirectional recurrent neural network* architecture. LambdaNet was another early approach, and it uses a *graph neural network* architecture. InCoder is a recent *large language model* that predicts arbitrary code completions, and while not trained specifically to predict type annotations, its “fill in the middle” capability makes it ideal for that task. All three models use training data from public code repositories.

We require a system that takes a JavaScript project as input and outputs a type-annotated TypeScript project. DeepTyper and LambdaNet output a probability distribution of types for each identifier, which we must then “weave” into the original JavaScript source to produce TypeScript; we describe this technique in Section 3.3. InCoder is a general-purpose, multi-language transformer, so we implemented a front end to use InCoder to predict type annotations and output TypeScript. Currently, our front end only supports type predictions for function parameters; we describe our implementation in Section 3.2.1.

Our approach can be adapted to work with any type prediction model. Older models may require some work to adapt their outputs, but our InCoder front end can easily be extended to support other fill-in-the-middle models, such as OpenAI’s model [4] and SantaCoder [5].

¹ The union type `number | string` produces a type error in the function.

² The correct, but awkward, type annotation is `{x?: number, y?: number}`, which declares `x` and `y` as optional properties. If `x` or `y` were required, the assignment on Line 4 would be a type error. Alternatively, `any` is valid, but unhelpful.

2.2.1 DeepTyper

DeepTyper [23] predicts types for variables, function parameters, and function results using a fixed vocabulary of types, i.e., it cannot predict types declared by the program under analysis unless those types were observed during training. DeepTyper treats type inference as a machine translation problem from one language (unannotated TypeScript) to another (annotated TypeScript). Specifically, it uses a model based on a *bidirectional recurrent neural network* architecture to translate a sequence of tokens into a sequence of types: for each identifier in the source program, DeepTyper returns a probability distribution of predicted types. Because the input token sequence is perfectly aligned with the output type sequence, this task can also be considered a sequence annotation task, where an output type is expected for every input token.³ However, this approach treats each input token as independent from the others, i.e., a source variable may be referenced multiple times and each occurrence may have a different type. To mitigate this, DeepTyper adds a consistency layer to the neural network, which encourages – but cannot enforce – the model to treat multiple occurrences of the same identifier as related.

DeepTyper’s dataset is based on the top 1,000 most starred TypeScript projects on GitHub, as of February 2018. After cleaning to remove large files (those with more than 5,000 tokens) and projects that contained only TypeScript declaration files, the dataset was left with 776 TypeScript projects (containing about 62,000 files and about 24 million tokens), which were randomly split into 80% (620 projects) training data, 10% (78 projects) validation data, and 10% (78 projects) test data. Further processing and cleaning of rare tokens resulted in a final vocabulary of 40,195 source tokens and 11,830 types.

The final training dataset contains both identifiers and types, where each identifier has an associated type annotation; this includes annotations inferred by the TypeScript compiler that were not manually annotated by a programmer. The testing dataset contains type annotations and no identifiers; specifically, the type annotations added by programmers are associated with their declaration sites, and all other sites are associated with “no-type.” As a result, DeepTyper’s predictions are evaluated against the handwritten type annotations, rather than all types in a project.

2.2.2 LambdaNet

Like DeepTyper, LambdaNet [50] predicts type annotations for variables, function parameters, and function returns: it takes an unannotated TypeScript program and outputs a probability distribution of predicted types for each declaration site. LambdaNet improves upon two limitations of DeepTyper. First, it predicts from an open vocabulary, beyond the types that were observed during training; i.e., it can predict user-defined types from within a project. Second, it only produces type predictions at declaration sites, rather than at every variable occurrence; in other words, multiple uses of the same variable will have a consistent type.

LambdaNet uses a *graph neural network* architecture and represents a source program as a so-called *type dependency graph*, which is computed from an intermediate representation of TypeScript that names each subexpression. The type dependency graph is a hypergraph that encodes program type variables as nodes, and relationships between those variables as labeled edges. By encoding type variables, LambdaNet makes a single prediction over all occurrences

³ The DeepTyper architecture must classify *every* input token, including ones where an output type does not make sense, such as `if`, `(`, `)`, and even whitespace. DeepTyper filters out these predictions, so a user will never observe these meaningless types.

37:6 Do Machine Learning Models Produce TypeScript Types That Type Check?

of a variable, rather than a prediction for each instance of a variable. Furthermore, the edges encode logical constraints and contextual hints. Logical constraints include subtyping and assignment relations, functions and calls, objects, and field accesses, while contextual hints include variable names and usages. Finally, LambdaNet uses a *pointer network* to predict type annotations.

LambdaNet’s dataset takes a similar approach to DeepTyper: they selected the 300 most popular TypeScript projects from GitHub that contained 500–10,000 lines of code, and had at least 10% of type annotations that referred to user-defined types. The dataset has about 1.2 million lines of code, and only 2.7% of the code is duplicated. The 300 projects were split into three sets: 200 (67%) for training data, 40 (13%) for validation data, and 60 (20%) for test data. The vocabulary was split into library types, which consist of the top 100 most common types in the training set, and user-defined types, which are all the classes and interfaces defined in source projects. Similar to DeepTyper, LambdaNet’s predictions are evaluated against the handwritten annotations that were added by programmers.

2.2.3 InCoder

InCoder [18] is a *large language model* (LLM) for generating arbitrary code that is trained with a *fill-in-the-middle* (FIM) objective on a corpus of several programming languages, including TypeScript.

InCoder’s corpus consists of permissively licensed, open-source code from GitHub and GitLab, as well as Q&A and comments from Stack Overflow. This raw data is filtered to exclude: (1) code that is duplicated; (2) code that is not written in one of 28 languages; (3) files that are extremely large or contain very few alphanumeric characters; (4) code that is likely to be compiler generated; and (5) certain code generation benchmarks. The result is about 159 GB of code, which is dominated by Python and JavaScript. TypeScript is approximately 4.5 GB of the training data.

We describe InCoder in more depth in Section 3.2.1, where we present what is necessary to use it as a type annotation prediction tool for TypeScript.

2.2.4 Evaluating on Accuracy

The main evaluation criteria for the type annotation prediction task is accuracy: what is the likelihood that a predicted type annotation is correct? Correct means the prediction *exactly* matches the ground truth, which is the handwritten type annotation at that location. Accuracy is typically measured as top- k accuracy, where a prediction is deemed correct if any of the top k most probable predictions is correct. For our evaluation, we would like a result that “just works,” i.e., a program that type checks. Therefore, we are only interested in top-1 accuracy, since we take the top guess as the only prediction.

DeepTyper’s test dataset makes up 10% (78 projects) of its original corpus and contains only the annotations that were manually added by programmers. Predictions are compared against this ground truth dataset, and DeepTyper reports a top-1 accuracy of 56.9%. Because DeepTyper may predict different types for multiple occurrences of the same variable, the authors also report an inconsistency metric: 15.4% of variables had multiple type predictions.

LambdaNet also compares its predictions against a ground truth of handwritten type annotations, but they use a different corpus and split 20% (60 projects) for the test dataset. LambdaNet can predict user-defined types, so the evaluation reports two sets of results: a top-1 accuracy of 75.6% when predicting only common library types, and a top-1 accuracy of 64.2% when predicting both library and user-defined types.

InCoder was not designed specifically to predict TypeScript type annotations, but the authors report an experiment to predict only the result types for Python functions. For this task, InCoder was evaluated on a test dataset of 469 functions, which was constructed from the CodeXGLUE dataset; InCoder achieved an accuracy of 58.1%.

However, we argue that accuracy is not the right metric for evaluating a type prediction model. As a first step, we would like to type check the TypeScript project. Additionally, when migrating a JavaScript project to TypeScript, there is frequently no ground truth of handwritten type annotations; instead, the ground truth is what the compiler accepts. This condition is much stronger than accuracy, as even a single, incorrect type annotation causes a package to fail to type check. On the other hand, less precise type annotations (e.g., `any`) and equivalent annotations (e.g., `number | string` vs. `string | number`) may be accepted, despite not matching the ground truth exactly.

In the next section, we describe our approach for type migration and evaluating type prediction models.

3 Approach

We take an end-to-end approach to type migration, starting from an untyped JavaScript project and finishing with a type-annotated TypeScript project that we try to type check. The first step, which is optional, is to convert from CommonJS modules to ECMAScript modules. Next, we invoke one of the type prediction models: DeepTyper and LambdaNet produce type predictions, while InCoder, with a front end we implemented, produces TypeScript. Because DeepTyper and LambdaNet do not produce TypeScript, we perform a step that we call *type weaving*, which combines type predictions with the original JavaScript source and outputs TypeScript. Finally, we run the TypeScript compiler to type check the now migrated TypeScript project.

3.1 CommonJS to ECMAScript Module Conversion

The first step is to convert projects from CommonJS module notation to ECMAScript module notation. This step is not necessary for type prediction, but is important for the type checking evaluation, as only ECMAScript modules preserve type information across module boundaries. Because we treat this step as optional, our dataset has two versions: the original projects, which may use CommonJS or ECMAScript modules, and a final version that only uses ECMAScript modules.

Most Node.js packages use the CommonJS module system, which was the original module system for Node.js and remains the default. Figures 2a and 2b show an example of the CommonJS module system, where files `a.js` and `b.js` implement separate modules. In this example, `a.js` sets the `foo` and `f` properties of the special `module.exports` object. Local variables like `x` are private and not exported. On Line 13, `b.js` uses the Node.js function `require` to load module `a.js` into the local variable `a`. As a result, `a` takes on the value of the `module.exports` object set by `a.js`, and both `foo` and `f` are available as properties of `a`.

ECMAScript 6 introduced a new module system, referred to as ECMAScript modules. Node.js supports ECMAScript modules when using the `.mjs` extension or setting a project-wide configuration in the `package.json` file. Figures 2c and 2d show the same program as before, but rewritten to use ECMAScript modules. In this example, `a.mjs` directly exports `foo` and `f`, rather than writing to a special `module.exports` object. Then, `b.mjs` directly imports the names with the `import` statement instead of loading an object.

37:8 Do Machine Learning Models Produce TypeScript Types That Type Check?

```
7 // a.js
8 var x = 2;    // private
9
10 module.exports.foo = 42;
11 module.exports.f = (i) => i+x;
```

(a) CommonJS: a.js exports foo and f.

```
12 // b.js
13 var a = require('./a.js');
14
15 console.log(a.foo); // 42
16 console.log(a.f(1)); // 3
```

(b) CommonJS: b.js imports the module a.js.

```
17 // a.mjs
18 var x = 2;    // private
19
20 export var foo = 42;
21 export var f = (i) => i+x;
```

(c) ECMAScript: a.mjs exports foo and f.

```
22 // b.mjs
23 import {foo,f} from './a.mjs';
24
25 console.log(foo); // 42
26 console.log(f(1)); // 3
```

(d) ECMAScript: b.mjs imports foo and f.

■ **Figure 2** An example comparing the CommonJS and ECMAScript module systems.

TypeScript supports both CommonJS and ECMAScript modules, depending on the project configuration. However, CommonJS modules in TypeScript are untyped; specifically, `require` is typed as a function that returns `any`. Therefore, even if a module has type annotations for the variables and functions it exports, those annotations are lost when the module is imported. On the other hand, with ECMAScript modules, the `import` statement preserves the type annotations of names it imports.

In order to make use of the most type information available, we would like to use ECMAScript modules in our evaluation. Therefore, we use the `cjs-to-es6` tool⁴ to transform our dataset to use ECMAScript modules. The conversion tool is not perfect, and in particular has difficulty when `require` is used to dynamically load a module. Some of these cases could be fixed manually, but many are genuine uses of dynamic loading in JavaScript.

3.2 Type Annotation Prediction

The next step is to invoke a deep learning model to predict type annotations for a JavaScript project. DeepTyper and LambdaNet require an additional step, which we call *type weaving*, to produce TypeScript, while InCoder, with our front end, outputs TypeScript directly.

We use the pretrained DeepTyper model available from its GitHub repository,⁵ which is not identical to the model used in the DeepTyper paper. DeepTyper reads in a JavaScript file, and for each identifier, predicts the top five most likely types, outputting the result in comma-separated values (CSV) format.

We use the pretrained LambdaNet model available from its GitHub repository,⁶ specifically the model that supports user-defined types. LambdaNet reads in a directory containing a JavaScript project, and predicts the top five most likely types for each variable and function declaration. We modify LambdaNet to output in CSV format.

DeepTyper predicts types for all identifiers in the program, including program locations that do not allow type annotations. Therefore, type weaving must also ensure that type annotations are applied correctly, i.e., only to variable declarations, function parameters, and function results. LambdaNet predicts types for variable and function declarations, and in

⁴ <https://github.com/nolanlawson/cjs-to-es6>

⁵ <https://github.com/DeepTyper/DeepTyper/tree/master/pretrained>

⁶ <https://github.com/MrVPlusOne/LambdaNet/tree/ICLR20>

```

27 function f(x) {
28   return x + 1;
29 }

```

(a) An example program.

```

30 function f(x: <|mask:0|>) {
31   return x + 1;
32 }<|mask:1|><|mask:0|>

```

(b) Preparing code for generation.

```

33 function f(x: <|mask:0|>) {
34   return x + 1;
35 }<|mask:1|><|mask:0|>
36 number, y: number<|endofmask|>

```

(c) InCoder often produces extra tokens after the type. Here it produces a new parameter that is not in the original program.

```

37 function f(x: number) {
38   return x + 1;
39 }

```

(d) We select a prefix of the generated program that is a syntactically valid TypeScript type.

■ **Figure 3** Generating types with InCoder.

the correct locations; however, type weaving is still required to produce TypeScript code. Our InCoder front end does not require type weaving, but only supports type predictions for function parameters. We use the pretrained InCoder model available from Hugging Face.⁷

3.2.1 InCoder

InCoder is trained to generate code in the middle of a program, conditioned on the surrounding code. To train on a single example (a file of code), the training procedure replaces a randomly selected contiguous span of tokens with a *mask sentinel* token. It appends the mask sentinel to the end of the example, followed by the tokens that were replaced and a special *end-of-mask* token. The model is then trained as a left-to-right language model. This approach generalizes to support several, non-overlapping masked spans, and its training examples have up to 256 randomly selected masked spans, though the majority have just a single masked span.

In principle one could give InCoder a program with up to 256 types to generate at once. However, we found that InCoder is more successful generating a single type at a time, and with a limited amount of context. To generate a type annotation with InCoder we (1) insert the mask sentinel token at the insertion point; (2) add the mask sentinel to the end of the file; (3) generate at the end of the file until the model produces the end-of-mask token; (4) move the generated text to the insertion point; and (5) remove all sentinels.

Figure 3 shows an example of generating a type annotation. However, a problem that we frequently encountered is that InCoder sometimes generates more than just a single type. For instance, Figure 3c shows an example where InCoder generates a new parameter that is not in the original program. The simplest approach is to reject this result and get InCoder to re-generate completions until it produces a type. However, we found that it is far more efficient to accept a prefix of the generated code if it is a syntactically valid type, which we check with a TypeScript parser in the generation loop.

⁷ <https://huggingface.co/facebook/incoder-6B>

3.3 Type Weaving

To produce type-annotated TypeScript code, we use a process we call *type weaving* to combine type predictions with the original JavaScript code. Type weaving takes two files as input: a JavaScript source file and an associated CSV file with type predictions. The type weaving program parses the JavaScript source into an abstract syntax tree (AST), and then traverses the AST and CSV files simultaneously, using the TypeScript compiler to insert type annotations into the program AST. Both DeepTyper and LambdaNet require type weaving, but their CSV files are in different formats. Our type weaving program can be extended to support custom CSV formats.

3.3.1 DeepTyper

Each row of a DeepTyper CSV file represents a lexical token from the source program. Rows with non-identifier tokens, such as keywords and symbols, contain two columns: the token text and the token type. Rows with identifiers contain columns for the token text, token type, as well as the top five most likely types and their probabilities.

The DeepTyper implementation has a few limitations that we handle during type weaving. First, the implementation uses regular expressions instead of a parser to tokenize JavaScript code. This results in some tokens that are missing or incorrectly classified as identifiers. Second, DeepTyper provides type predictions for every occurrence of an identifier, so we must use only the predictions for declarations. Finally, DeepTyper often predicts `complex` as a type; we do not believe it refers to a complex number type, so we replace it with `any`.

Our type weaving algorithm works as follows: as it traverses the program AST, if it encounters a declaration node, it queries the CSV file for a type prediction. However, the DeepTyper format does not record source location information, and the token classification is brittle, so it is not straightforward to identify which rows are actually declarations and which rows should be skipped. Our algorithm searches the CSV file for a short sequence of rows that corresponds to the declaration node in the AST. This algorithm works well in practice, but does not handle optional parameters or statements that declare multiple variables.

3.3.2 LambdaNet

For each declaration, LambdaNet prints the source location of the identifier (start line, start column, end line, and end column), followed by the top five most likely types and their probabilities. We modified LambdaNet to output in CSV format.

We observed that LambdaNet frequently predicts the following types: `Number`, `Boolean`, `String`, `Object`, and `Void`. The first four are valid TypeScript types, but are non-primitive boxed types distinct from `number`, `boolean`, `string`, and `object`. The TypeScript documentation strongly recommends using the lowercase type names,⁸ so we normalize those types during type weaving. Furthermore, `Void` is not a valid type, so we instead use `void`. Finally, LambdaNet does not support generic types, but will predict them without type arguments, which is not valid in TypeScript. While we cannot fix every generic type, we normalize `Array` to `any[]`, which is shorthand for `Array<any>`.

As our type weaving program traverses the program AST, if it encounters a declaration node, it computes the node's source location information, and uses that to query the CSV file for a type prediction. However, the type annotation cannot be applied directly to the

⁸ <https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html#number-string-boolean-symbol-and-object>

■ **Table 1** Summary of dataset categories: number of packages, files, and lines of code.

Dataset category	Packages	Files	Lines of code
DefinitelyTyped, no deps	286	2,692	123,157
DefinitelyTyped, with deps	85	671	63,057
Never typed, no deps	102	255	20,729
Never typed, with deps	40	544	19,189
Overall	513	4,162	226,132

declaration node, as this modifies the AST and invalidates source location information. Therefore, type weaving for LambdaNet occurs in two phases. In the first phase, the traversal does not modify the AST, but saves the declaration node and type prediction in a map. Then, in the second phase, type weaving iterates over the map and updates the AST.

3.4 Type Checking

In the final step, we run the TypeScript compiler to type check the migrated projects. We run the compiler on each project, providing all the TypeScript input files as arguments, and setting the following compiler flags:

```
--noEmit Type check only, do not emit JavaScript
--esModuleInterop Improve handling of CommonJS and ECMAScript modules
--moduleResolution node Explicitly set the module resolution strategy to Node.js
--target es6 Enable ECMAScript 6 features, which are used by some packages
--lib es2021,dom Include ECMAScript 2021 library definitions and browser DOM definitions
```

We do not set the `--strict` flag, allowing the type checker to be more lenient in certain situations, which we expect to already be a significant challenge for automated type migration. Furthermore, we ensure that package dependencies are properly included in our dataset so that the compiler can resolve them.

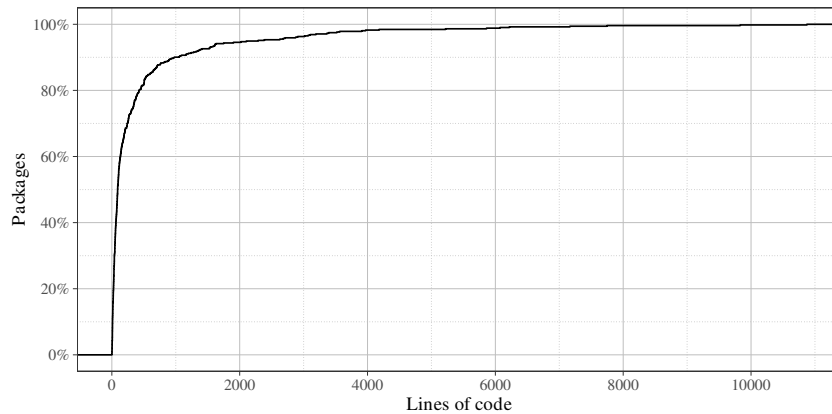
4 Evaluation

4.1 Dataset

Our dataset for evaluation consists of 513 JavaScript packages. To build this dataset, we start from the top 1,000 most downloaded packages from the npm Registry (as of August 2021) and narrow and clean as follows:

1. We add any transitive dependencies that are not in the original set of packages, to ensure that the dataset is closed.
2. We try to fetch the original source code of every package, and eliminate any package where this is not possible (e.g., the package did not provide a repository URL or was deleted from GitHub). Fetching the source helps ensure we are working with original code, and not compiled or “minified” JavaScript.
3. We remove packages that were built from a “monorepo,” i.e., a single repository containing multiple packages that are published separately. For example, the Babel JavaScript compiler has over 100 separate packages, but all share the same monorepo; fetching each source package meant downloading the entire monorepo multiple times and including unnecessary packages.

37:12 Do Machine Learning Models Produce TypeScript Types That Type Check?



■ **Figure 4** Empirical cumulative distribution function of lines of code per package, over all datasets. The x -axis shows lines of code and the y -axis shows the proportion of packages with fewer than x lines of code.

4. We remove packages that were not implemented in JavaScript, do not contain code, or have more than 10,000 lines of code. The size limit helps us avoid timeouts, and mostly excludes large toolchains and standard libraries, such as the TypeScript compiler and `core-js` standard library.
5. We remove testing code from every package. Tests frequently require extra dependencies, and different frameworks set up the test environment in different ways, which makes large-scale evaluation harder. To remove testing code, we deleted directories named `test`, `tests`, `__tests__`, or `spec`, and files named `test.js`, `tests.js`, `test-*.js`, `*-test.js`, `*.test.js`, or `*.spec.js`.
6. Finally, we ensure that every package has *no dependencies*, or that *all its dependencies are typed*, meaning the dependencies have TypeScript type declaration (`.d.ts`) files available. (We do not require that *packages* are typed, but only that their *dependencies* are.) This requirement is necessary because a JavaScript package can only be imported into a TypeScript project if its interface has TypeScript type declarations. The DefinitelyTyped repository⁹ contains interface type declarations for many popular JavaScript packages, and a handful of packages include their own. We download type declarations of project dependencies and include them in our dataset for evaluation purposes – they are not used for type prediction.

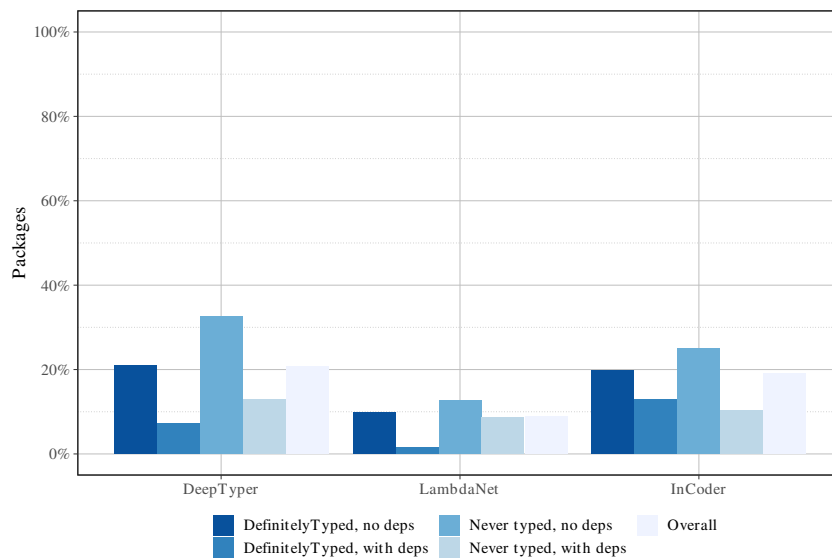
After filtering and cleaning our dataset, we classify every package with two criteria: (1) whether the package has type declarations available; and (2) whether the package has dependencies.

If a package has type declarations available, we say it is “DefinitelyTyped” and we can use its type annotations as ground truth in our evaluation. (However, there is evidence that some of these type annotations are incorrect [16, 28, 29, 51].) Otherwise, we use the term “never typed”: these packages *have never been type annotated* and thus no ground truth exists, so machine learning models have never been evaluated on these packages before. If a package has dependencies, we classify it as “with deps” (and from our filtering, we know that every dependency is typed); otherwise, we classify the package as “no deps.” Thus, there are four dataset categories; we list them in Table 1 along with the number of packages, files, and lines of code for each category.

⁹ <https://github.com/DefinitelyTyped/DefinitelyTyped/>

■ **Table 2** Number and percentage of packages that type check.

Dataset category	DeepTyper			LambdaNet			InCoder		
	✓	Total	%	✓	Total	%	✓	Total	%
DefinitelyTyped, no deps	54	257	21.0	24	247	9.7	55	277	19.9
DefinitelyTyped, with deps	5	69	7.2	1	70	1.4	10	77	13.0
Never typed, no deps	31	95	32.6	11	87	12.6	25	100	25.0
Never typed, with deps	5	39	12.8	3	35	8.6	4	39	10.3
Overall	95	460	20.7	39	439	8.9	94	493	19.1



■ **Figure 5** Percentage of packages that type check.

Figure 4 is an empirical cumulative distribution function of the lines of code per package: the x -axis shows lines of code in a package and the y -axis shows the proportion of packages with fewer than x lines of codes. From the graph, we observe that approximately 90% of packages have fewer than 1,000 lines of code, and approximately 95% of packages have fewer than 2,000 lines of code.

4.2 Success Rate of Type Checking

Do Migrated Packages Type Check?

We first ask if entire packages type check after automated migration from JavaScript to TypeScript. However, not all packages successfully translate to TypeScript with every type migration tool; some packages cause the type migration tool to time out or error. Thus, we report the success rate of type checking as a fraction of the packages that successfully translate to TypeScript.

Table 2 and Figure 5 show the fraction of packages that type check with each tool. We observe that DeepTyper and InCoder perform similarly (20% success rate), and LambdaNet performs worse (9% success rate). Across all tools, packages without dependencies type check at a higher rate than packages with dependencies.

37:14 Do Machine Learning Models Produce TypeScript Types That Type Check?

These package-level type checking results are disappointing – but this is a very high standard to meet. Even a single incorrect type annotation causes the entire package to fail. Therefore, we next consider a finer-grained metric that is still useful.

How Many Files are Error Free?

As an alternate measure, we look at the percentage of *files with no compilation errors*. Instead of a binary pass/fail outcome, this gives us a more fine-grained result for a package. We motivate this metric by observing that TypeScript files are modules with explicit imports and exports. If a file type checks without errors, then it is using all of its internal and imported types consistently. Thus, when triaging type errors, a programmer may (temporarily) set these files aside and focus on the files with compilation errors. However, the programmer may later need to return to a file with no type errors and adjust its type annotations, for example, if a consumer of that file expects a different interface. We give examples of this in our case studies, specifically Section 4.5.2 and Section 4.5.3.

Table 3 and Figure 6 present the fraction of files with no compilation errors. The results are more encouraging: using InCoder, 69% of files are error free. With these results, it is not clear that packages without dependencies outperform packages with dependencies. Finally, in Figure 7 we calculate the percentage of error-free files for each package, and plot histograms of the distribution. Across all tools, most packages have type errors in most or all files.

What Percentage of Type Annotations Are Trivial?

Next, we examine what percentage of type annotations, *within the error-free files*, are trivial, i.e., what percentage are `any`, `any[]` (array of anys), or `Function` (function that accepts any arguments and returns anything). These annotations can hide type errors and allow more code to type check; however, they provide little value to the programmer.

Figure 8 shows the percentage of trivial type annotations within error-free files. DeepTyper produces the most (about 60%), LambdaNet produces the least (about 25%), while InCoder is in between (about 40%).

Comparing to the percentage of files with no compilation errors (Figure 6), DeepTyper produces more type-correct code than LambdaNet, but it also generates more trivial type annotations. InCoder produces the most type-correct code, while generating a moderate percentage of trivial type annotations.

Do Migrated Types Match Human-Written Types (When Available)?

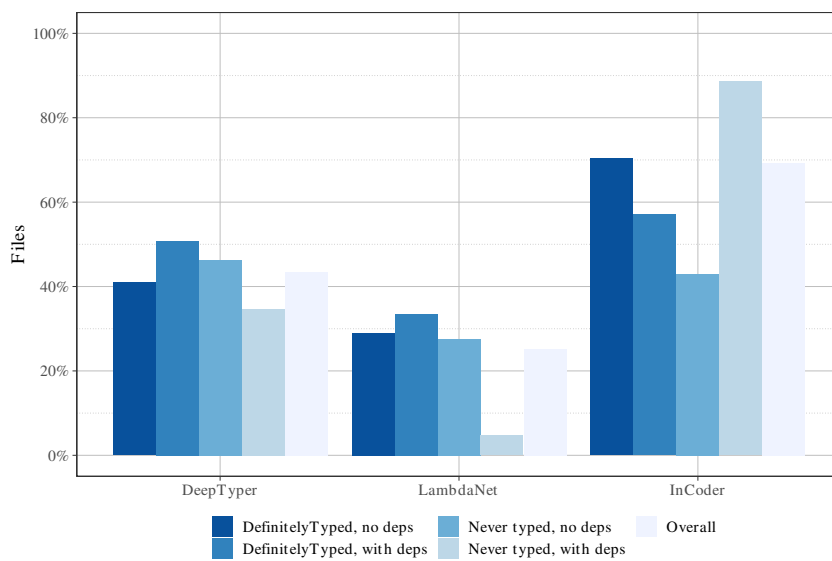
Since our dataset is constructed from JavaScript packages instead of TypeScript packages, we do not have fully type-annotated files as our ground truth; instead, we use declaration files provided by the DefinitelyTyped repository or package author. We configure the TypeScript compiler to emit declarations during type checking, which it can do even if the whole package does not type check. Thus, we can compare handwritten, ground truth declarations against declarations generated from migrated packages.

We extract function signatures from declaration files and only compare a signature if it is in both the ground truth and generated declaration. We compare the function parameter types and return types one-to-one, ignoring modifiers (e.g., `readonly`), and we require an exact string match (i.e. `string | number` and `number | string` are considered different types). Following the literature, we skip a comparison if the ground truth is the `any` annotation.

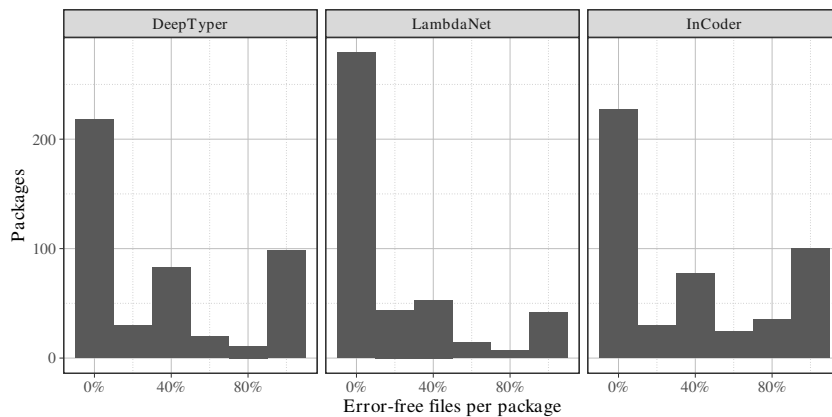
Our results are presented in Table 4 and Figure 9. We observe that accuracy is better for packages without dependencies. Additionally, our results follow the same pattern in prior work, where LambdaNet has better accuracy than DeepTyper, despite performing worse in our other metrics.

■ **Table 3** Number and percentage of files with no compilation errors.

Dataset category	DeepTyper			LambdaNet			InCoder		
	✓	Total	%	✓	Total	%	✓	Total	%
DefinitelyTyped, no deps	414	1,010	41.0	474	1,638	28.9	1,689	2,401	70.3
DefinitelyTyped, with deps	195	384	50.8	169	504	33.5	312	547	57.0
Never typed, no deps	95	205	46.3	63	229	27.5	101	235	43.0
Never typed, with deps	42	121	34.7	25	534	4.7	467	527	88.6
Overall	746	1,720	43.4	731	2,905	25.2	2,569	3,710	69.2

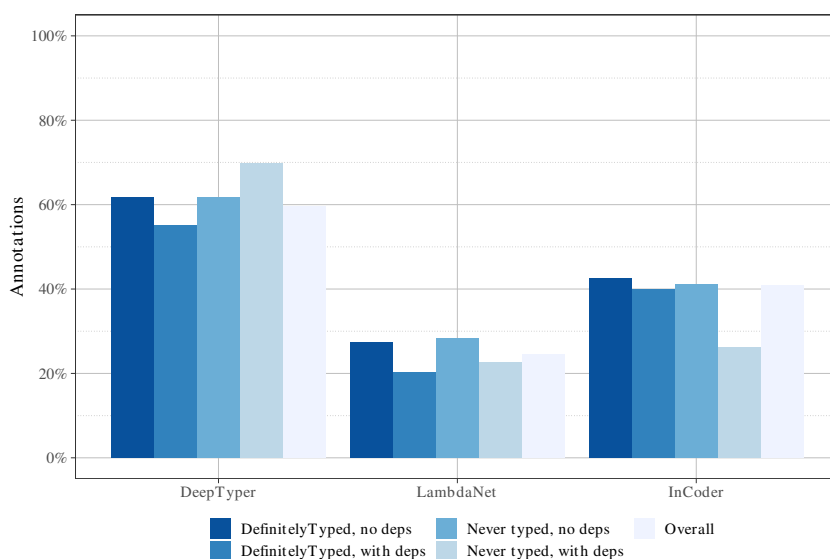


■ **Figure 6** Percentage of files with no compilation errors.



■ **Figure 7** Number of packages vs. percentage of error-free files per package.

37:16 Do Machine Learning Models Produce TypeScript Types That Type Check?



■ **Figure 8** Percentage of annotations that are `any`, `any[]`, or `Function`, in files with no errors.

■ **Table 4** Accuracy of type annotations, compared to non-`any` ground truth.

Dataset category	DeepTyper			LambdaNet			InCoder		
	✓	Total	%	✓	Total	%	✓	Total	%
DefinitelyTyped, no deps	90	241	37.3	116	259	44.8	40	123	32.5
DefinitelyTyped, with deps	27	123	22.0	41	119	34.5	11	64	17.2
Overall	117	364	32.1	157	378	41.5	51	187	27.3

How Many Errors Occur in Each Package?

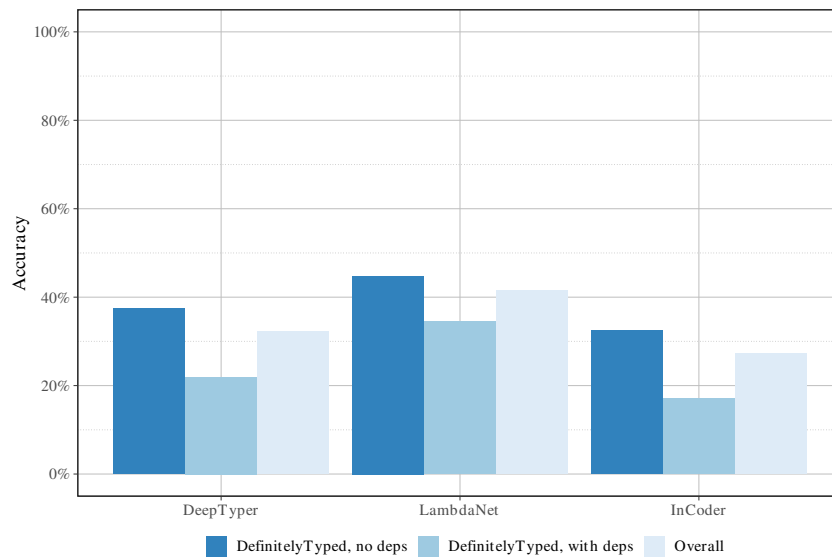
Figure 10 shows an empirical cumulative distribution function of errors: the x -axis shows the number of errors and the y -axis shows the proportion of packages with fewer than x errors. For example, when migrating the “DefinitelyTyped, with deps” dataset with LambdaNet, approximately 80% of packages have fewer than 250 errors each. Additionally, all of DeepTyper’s packages and almost all of InCoder’s packages have fewer than 500 errors each.

4.3 Error Analysis

We now consider the kinds of errors that arise during migration. Every TypeScript compiler error has an associated code,¹⁰ making categorization straightforward. Figure 11 summarizes the top 10 most common errors and Table 5 provides the corresponding messages.

Most of the errors relate to types. These errors are the following: a property not existing on a type (TS2339 and TS2551); an assignment with mismatched types (TS2322); a function call with mismatched parameter and argument types (TS2345); calling a function that was assigned a non-function type annotation (TS2349); and a conditional that compares values from different types (TS2367). TS2314 refers to a generic type that was not provided type arguments; this is caused by DeepTyper and LambdaNet not fully supporting generic types.

¹⁰<https://github.com/Microsoft/TypeScript/blob/v4.9.3/src/compiler/diagnosticMessages.json>



■ **Figure 9** Accuracy of type annotations, compared to non-any ground truth.

The remaining errors are not directly related to types. TS2304 refers to an unknown name, which may not necessarily be a type. TS2554 is emitted because TypeScript requires the number of call arguments to match the number of function parameters, but JavaScript does not. TS2339 includes cases where an empty object is initialized by setting its properties, but TypeScript requires that the object’s properties are declared in its type. Finally, TS2307 indicates that the ECMAScript module conversion produced incorrect code.

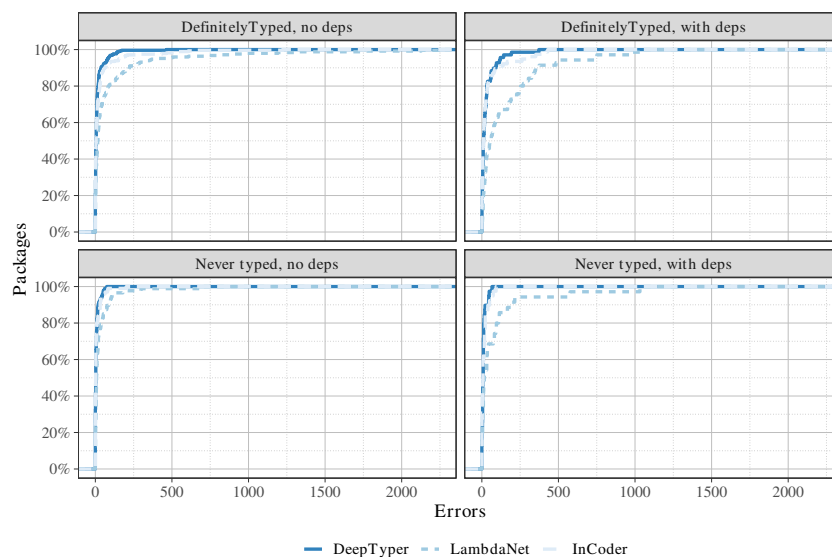
4.4 ECMAScript Module Conversion

Recall that we described an optional step before running the evaluation: converting packages to use ECMAScript modules. In this section, we re-run our evaluation – generating types, weaving types, and type checking – to compare the results before and after the conversion step. Specifically, we examine the percentage of packages that type check (Table 6), the percentage of files with no errors (Table 7), and accuracy (Table 8). However, this is not a direct comparison between CommonJS and ECMAScript modules, as some of the original packages were already using ECMAScript modules. Furthermore, the conversion affected a small handful of packages: some packages successfully migrated to TypeScript after the conversion but failed before, and the inverse was true for other packages.

From Table 6, we observe that the ECMAScript module conversion makes fewer packages type check for DeepTyper and InCoder, but slightly improves the results for LambdaNet. In Figure 12, we examine packages that type checked before or after the ECMAScript module conversion; packages that never type checked were excluded. In general, if a package type checked before the conversion, it likely type checked after the conversion. However, if a package failed to type check before the conversion, it was unlikely to type check afterwards; in fact, this never happened for a package with dependencies.

Table 7 compares the percentage of files with no compilation errors. The conversion improves the results for LambdaNet and InCoder, but makes the results worse for DeepTyper. One dramatic result is the change for InCoder and the “never typed, with deps” dataset, where the ECMAScript module conversion results in 89% of files type checking, when it was only

37:18 Do Machine Learning Models Produce TypeScript Types That Type Check?



■ **Figure 10** Empirical cumulative distribution function of errors. The x -axis shows the number of errors and the y -axis shows the proportion of packages with fewer than x errors.

11% before. The difference is caused by a single package, `regenerate-unicode-properties`, which has over 400 files. With CommonJS modules, each file produces an error; however, with ECMAScript modules, those files type check successfully.

Finally, Table 8 compares the accuracy of type annotations, before and after the ECMAScript module conversion. Recall that for accuracy, we compare type annotations against the ground truth of handwritten TypeScript declaration files; these are the “DefinitelyTyped” datasets. Accuracy improves for DeepTyper and LambdaNet, but worsens slightly for InCoder.

4.5 Case Studies

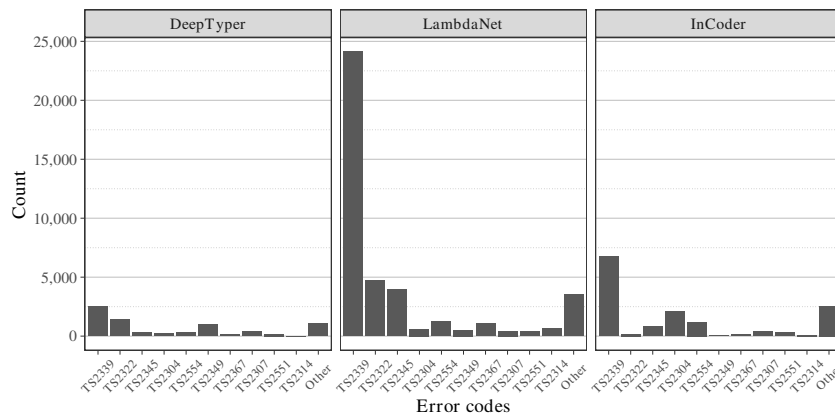
In this section, we examine how the models performed on four packages, whether the packages type check, and what steps are left to migrate the packages to TypeScript.

4.5.1 Error Message Does Not Refer to Incorrect Type Annotation

`decamelize` is a package for converting strings in camel case to lowercase.¹¹ It is in the “DefinitelyTyped, no deps” dataset, as it ships with a `.d.ts` declaration file and has no dependencies. Figure 13 shows a simplified version of the function `handlePreserveConsecutiveUppercase`. This function is not exported, thus there are no programmer-written type annotations. Line 42 uses JavaScript’s arrow function notation to define a function that takes two arguments, and assigns it to the constant on Line 41. Elsewhere in the package (Line 51 in the listing), the helper function is called with `str` and `sep` string arguments.

A programmer inspecting the function can reason that Line 44 is a call to a string method that uses a regular expression on Line 45 to replace text in `decamelized` with the result on Line 46, where `separator` is concatenated with the regular expression match. Therefore, both the `decamelized` and `separator` parameters on Line 42 should be annotated as `string`.

¹¹<https://www.npmjs.com/package/decamelize>



■ **Figure 11** Distribution of the top 10 most common error codes, over all datasets.

The DeepTyper solution is listed on Line 54: it correctly annotates both function parameters as `string`, but incorrectly annotates `handlePreserveConsecutiveUppercase` as `string`. The compiler emits errors for Lines 51 and 55, because Line 51 is attempting to call a non-function, and Line 55 is attempting to assign a function to a non-function variable. However, the fix must be applied to the annotation on line Line 54.

4.5.2 Incorrect Type Annotation Can Type Check Successfully

The LambdaNet solution on Line 58 correctly annotates `handlePreserveConsecutiveUppercase` as `Function`, but it incorrectly annotates the `separator` parameter on Line 59 as `number`. We would expect the compiler to emit a type error, since Line 51 calls the function with string arguments. However, the code type checks successfully, because the generic `Function` type on Line 58 accepts any number of arguments of any type. The `Function` type annotation is similar to `any`, in that it enables more code to type check, but at the cost of fewer type guarantees.

Another example of this problem is the `ieee754` package, which reads and writes floating point numbers to and from buffers.¹² It is categorized as “DefinitelyTyped, no deps,” since it provides a `.d.ts` declaration file and has no dependencies. Figure 14 shows the original declaration for the `write` function on Line 61, and the handwritten, ground truth signature on Line 65.

Consider the DeepTyper solution: the compiler emits an error on Line 70, because a function is being assigned to a variable of type `void`. However, even if that error is fixed, there is another, more subtle error not detected by the compiler: the `isLE` parameter on Line 71 is incorrectly annotated as `number`, not `boolean`. Because this is compatible with the body of the function, there is no error. (The `Buffer` annotation is valid, despite not matching the ground truth `Uint8Array`, because `Buffer` is defined by the Node.js standard library as a subtype of `Uint8Array`.)

The InCoder solution on Line 75 type checks successfully. It also uses the `Buffer` type for `buffer`, and it uses `any` instead of `number` for the `value` parameter on Line 76. The `any` annotation may cause run-time errors if the function is called with arguments of the wrong type.

¹²<https://www.npmjs.com/package/ieee754>

■ **Table 5** The top 10 most common error codes.

Error code	Message	DeepTyper	LambdaNet	InCoder
TS2339	Property '{0}' does not exist on type '{1}'.	2,510	24,123	6,742
TS2322	Type '{0}' is not assignable to type '{1}'.	1,429	4,709	176
TS2345	Argument of type '{0}' is not assignable to parameter of type '{1}'.	304	3,930	828
TS2304	Cannot find name '{0}'.	217	598	2,094
TS2554	Expected {0} arguments, but got {1}.	330	1,234	1,200
TS2349	This expression is not callable.	977	529	26
TS2367	This condition will always return '{0}' since the types '{1}' and '{2}' have no overlap.	145	1,046	110
TS2307	Cannot find module '{0}' or its corresponding type declarations.	375	432	371
TS2551	Property '{0}' does not exist on type '{1}'. Did you mean '{2}'?	187	389	296
TS2314	Generic type '{0}' requires {1} type argument(s).	1	630	95
Other		1,089	3,528	2,557
Total		7,564	41,148	14,495

4.5.3 Run-Time Type Assertions

The `@gar/promisify` package,¹³ simplified and shown in Figure 15, is another example where a program type checks, but is incorrect. The example exports a function that takes an argument `thingToPromisify`, type annotated as `string` by LambdaNet. Line 80 performs a run-time type check with the `typeof` operator. This ensures that `thingToPromisify` is a function on Line 81, which is what the `promisify` function, defined by Node.js, expects. If `thingToPromisify` is not a function, the exception on Line 83 is thrown.

The example type checks successfully, because the TypeScript compiler treats the `typeof` check as a type guard, and reasons that on Line 81, the `thingToPromisify` variable has been narrowed¹⁴ to a more specific type. However, because `thingToPromisify` is annotated as `string`, the type guard always returns false. Therefore, Line 81 is actually unreachable, so the exception on Line 83 is always thrown.

4.5.4 Variable Used as Two Different Types

The example in Figure 16 is adapted from the `array-unique` package.¹⁵ The example contains two `for` loops: a traditional, counter-based `for` loop on Line 90, and a `for...in` loop on Line 92 that iterates over all enumerable string properties of an object. Both loops share the same loop variable, `i`, defined on Line 88 and annotated as `number` by LambdaNet.

The use of `i` on Line 92 causes a type error, as `for...in` loops require the loop variable to be `string`. However, changing the annotation on Line 88 to `string` causes a type error on Line 90, as counter-based `for` loops require the loop variable to be `number`. One solution is to

¹³<https://www.npmjs.com/package/@gar/promisify>

¹⁴<https://www.typescriptlang.org/docs/handbook/2/narrowing.html>

¹⁵<https://www.npmjs.com/package/array-unique>

■ **Table 6** Percentage of packages that type check, before and after ECMAScript module conversion.

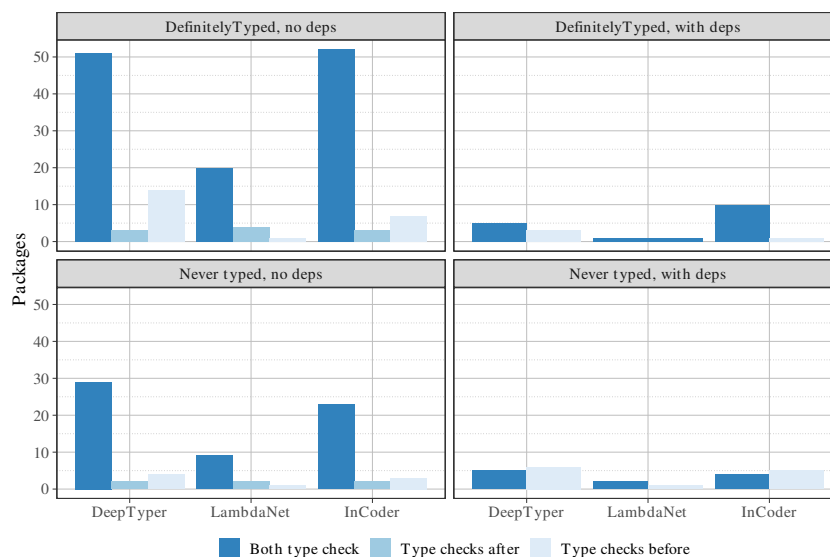
Dataset category	DeepTyper		LambdaNet		InCoder	
	Before	After	Before	After	Before	After
DefinitelyTyped, no deps	25.3	21.0	8.8	9.7	21.3	19.9
DefinitelyTyped, with deps	11.6	7.2	2.9	1.4	14.3	13.0
Never typed, no deps	34.7	32.6	11.5	12.6	26.0	25.0
Never typed, with deps	28.2	12.8	8.8	8.6	23.1	10.3
Overall	25.4	20.7	8.4	8.9	21.3	19.1

■ **Table 7** Percentage of files with no compilation errors, before and after ECMAScript module conversion.

Dataset category	DeepTyper		LambdaNet		InCoder	
	Before	After	Before	After	Before	After
DefinitelyTyped, no deps	45.3	41.0	27.9	28.9	68.3	70.3
DefinitelyTyped, with deps	50.0	50.8	24.3	33.5	51.6	57.0
Never typed, no deps	48.8	46.3	24.9	27.5	42.6	43.0
Never typed, with deps	64.5	34.7	8.6	4.7	11.2	88.6
Overall	48.1	43.4	23.5	25.2	56.1	69.2

■ **Table 8** Accuracy of type annotations, before and after ECMAScript module conversion.

Dataset category	DeepTyper		LambdaNet		InCoder	
	Before	After	Before	After	Before	After
DefinitelyTyped, no deps	35.2	37.3	44.4	44.8	34.0	32.5
DefinitelyTyped, with deps	19.8	22.0	27.0	34.5	19.2	17.2
Overall	28.7	32.1	38.9	41.5	29.1	27.3



■ **Figure 12** Packages that type check before or after ECMAScript module conversion.

37:22 Do Machine Learning Models Produce TypeScript Types That Type Check?

```
40 // Original
41 const handlePreserveConsecutiveUppercase =
42   (decamelized, separator) => {
43     // code omitted and simplified...
44     return decamelized.replace(
45       /([A-Z]+)([A-Z][a-z]+)/gu,
46       (_, $1, $2) => $1 + separator + $2.toLowerCase(),
47     );
48   }
49
50 // Elsewhere in the package; str and sep are both strings
51 return handlePreserveConsecutiveUppercase(str, sep);
52
53 // DeepTyper solution
54 const handlePreserveConsecutiveUppercase: string =
55   (decamelized: string, separator: string) => { ... }
56
57 // LambdaNet solution
58 const handlePreserveConsecutiveUppercase: Function =
59   (decamelized: string, separator: number) => { ... }
```

■ **Figure 13** The `handlePreserveConsecutiveUppercase` function adapted from the `decamelize` package. The `DeepTyper` and `LambdaNet` solutions are also shown.

use the `any` annotation, and another is to use the union type `number | string`. Ultimately, the correct solution is to define separate loop variables; this example highlights that code written in JavaScript may need to be refactored for TypeScript.

5 Discussion

How should type prediction models be evaluated? Prior work has used accuracy to evaluate type prediction models, but in this paper, we argue that we should instead type check the generated code. However, we acknowledge that our proposed metric also has limitations: code may type check with trivial annotations (e.g., `any` or `Function`) that provide little benefit to the programmer. Furthermore, type correctness does not necessarily mean the type annotations are correct: `any` can hide type errors that are only encountered at run time.

We do not claim that our metric is the final word on evaluating type prediction models, but we believe it is an improvement over accuracy. We hope this paper can spark a discussion on how machine learning for type migration should be evaluated.

Can slightly wrong type annotations be useful? A type prediction model may suggest types that are slightly wrong and easily fixable by a programmer, but fail to type check. However, a tool that produces hundreds or thousands of slightly wrong type annotations would overwhelm the programmer, and we believe it is important to build tools that try to produce fewer errors. On the other hand, slightly wrong type annotations may still provide value, but we would need to define what “slightly wrong” means and how to measure it. Without a tool like `TYPEWEAVER`, which weaves type annotations into code and type checks the result, it would not be possible to ask these questions.

Should we evaluate on JavaScript or TypeScript programs? We choose to evaluate on JavaScript programs, so our dataset deviates from prior work, which only considered TypeScript. Our motivating problem is not to recover type annotations for TypeScript


```

60 // Original
61 export const write =
62   function (buffer, value, offset, isLE, mLen, nBytes) { ... }
63
64 // Ground truth signature
65 export function write(
66   buffer: Uint8Array, value: number, offset: number, isLE: boolean,
67   mLen: number, nBytes: number): void;
68
69 // DeepTyper solution
70 export const write: void = function (
71   buffer: Buffer, value: number, offset: number, isLE: number,
72   mLen: number, nBytes: number) { ... }
73
74 // InCoder solution
75 export const write = function (
76   buffer: Buffer, value: any, offset: number, isLE: boolean,
77   mLen: number, nBytes: number) { ... }

```

■ **Figure 14** The `write` function adapted from the `ieee754` package. The ground truth signature is also shown, along with the DeepTyper and InCoder solutions.

```

78 // LambdaNet solution
79 export default function (thingToPromisify: string) {
80   if (typeof thingToPromisify === 'function') {
81     return promisify(thingToPromisify)
82   }
83   throw new TypeError('Can only promisify functions or objects')
84 };

```

■ **Figure 15** The LambdaNet solution for a function adapted from the `@gar/promisify` package.

programs that already type check, but to migrate untyped JavaScript programs to type-annotated TypeScript. For this problem, type prediction on its own is not enough, and other steps and further refactoring may be required.

Our methodology makes it possible to evaluate performance on code without known type annotations, i.e., code that has never been typed before. In contrast, prior work required the benchmarks to have ground truth type annotations. Our approach also reduces the likelihood of training data leaking into the test set.

However, there may be scenarios where a type prediction model is used to generate type annotations for a partially annotated TypeScript project. In these situations, type migration would likely not require additional refactoring steps.

Can we fully automate type migration? Our results show that automatically predicting type annotations is a challenging task and much work remains to be done. Furthermore, migrating JavaScript to TypeScript involves more than just adding type annotations: the two languages are different and some refactoring may be required. The models we evaluate in this paper do not refactor code, and we believe it is unlikely for automated type migration to be perfect. Thus, some manual refactoring will always be necessary for certain kinds of code, but we hope that tools can reduce the overall burden on programmers.

```

85 export default function(arr: any[]) {
86   var len: number = arr.length;
87   var o: object = {};
88   var i: number;
89
90   for (i = 0; i < len; i += 1) { ... }
91
92   for (i in o) { ... }
93 };

```

■ **Figure 16** The LambdaNet solution for a function adapted from the `array-unique` package.

6 Related Work

There are many constraint-based approaches to type migration for the gradually typed lambda calculus (GTLC) and some modest extensions. The earliest approach was a variation of unification-based type inference [47], and more recent work uses a wide range of techniques [9, 12, 21, 34, 35, 40]. Since these approaches are based on programming language semantics, they produce sound results, which is their key advantage over learning-based approaches. However, these would require significant work to scale to complex programming languages such as JavaScript.

There are also several constraint-based approaches to type inference for larger languages. Anderson et al. [2] presents type inference for a small fragment of JavaScript, but is not designed for gradual typing. Rastogi et al. [42] infer gradual types for ActionScript to improve performance. More recently, Chandra et al. [13] infer types for JavaScript programs with the goal of compiling them to run efficiently on low-powered devices; their approach is not gradual by design and deliberately rejects certain programs. DRuby [20] infers types for Ruby and treats type annotations in a novel way: inference assumes that annotations are correct, and defers checking them to runtime.

Although this paper focuses on type migration for TypeScript, there are several other gradual type systems for JavaScript [15, 22, 30, 49]. These languages do not have support for type inference and do not provide tools for type migration. Instead, like Typed Racket [48], they require programmers to manually migrate their code to add types. However, there are tools that use dynamic profiles to infer types for these type systems [1, 19, 45].

Even when constraint-based type inference succeeds in a gradually typed language, it can fail to produce the kinds of types that programmers write, e.g., named types, instead of the most general structural type for every annotation. Soft Scheme [10] infers types for Scheme programs, but Flanagan [17, p. 41] reports that it produces unintuitive types. For Ruby, InferDL [27] uses hand-coded heuristics to infer more natural types, and SimTyper [26] uses machine learning to predict equalities between structural types and more natural types.

LambdaNet [50] and DeepTyper [23] are two different approaches for predicting types for TypeScript and JavaScript programs. This paper evaluates using both of them in its type migration pipeline. We discuss them at length in Sections 2.2.1 and 2.2.2. NL2Type [32] is another system for predicting JavaScript types that improves on DeepTyper.

There are also type prediction systems for Python. TypeWriter [41] is notable because it also asks if the resulting Python program type checks. If it does not, it searches its solution space for an alternative typing. A distinction between Python type systems and TypeScript is that Python code is predominantly nominally typed: the type of a variable is either a builtin type or a class, whereas TypeScript uses structural types.

DiverseTyper [24] is a recently published work that predicts both built-in and user-defined types for TypeScript and achieves state-of-the-art accuracy on type prediction. DiverseTyper builds on TypeBert [25], which trains a BERT-based model to predict types. Although this paper does not evaluate these models, they are most closely related to InCoder [18], which is a general-purpose code generation model that we do evaluate.

7 Conclusion

In this paper, we set out to answer the question: *do deep-learning-based type annotation prediction models produce TypeScript types that type check?* To answer this question, we build TYPEWEAVER, a type migration tool that automatically converts JavaScript projects into TypeScript. TYPEWEAVER uses a type annotation prediction model, but does the work of “weaving” predicted types into JavaScript code. It also automates other steps, such as converting JavaScript projects to ECMAScript module notation. Finally, TYPEWEAVER runs the TypeScript compiler to type check the generated code. TYPEWEAVER is designed so that any type prediction model can be plugged in, and we use three very different models: DeepTyper, LambdaNet, and InCoder.

In addition to building TYPEWEAVER, we also present a dataset of 513 widely used JavaScript packages that are suitable for type migration. Every package in our dataset has typed dependencies and many of them have never been typed before. With this dataset, we evaluate TYPEWEAVER with all three type prediction models.

The results are mixed. If we ask, “How many packages type check when migrated to TypeScript?” we find that most packages have some type errors. However, we also ask, “How many files are error free?” and the result is more promising. We find that most files have no type errors, which means that programmers performing type migration can focus their attention on a smaller number of files.

Our case studies highlight two insights: (1) certain patterns in JavaScript do not make sense in TypeScript, so a migration may require manual rewriting of the code; and (2) there are cases where programs successfully type check but still have run-time errors.

We believe that currently, while type prediction cannot always reliably migrate JavaScript to TypeScript, it can still be a powerful tool.

Future Work. There are several directions we would like to explore in future work. First, we would like to improve dataset quality. We observed projects that were trivially typable: there were few declarations to annotate, or the annotations were mostly primitive types, so those projects often type checked successfully. Second, we are interested in exploring different evaluation criteria for type prediction models. We believe that type checking the output of these models is only the first step, and that it may be necessary to evaluate the run-time behavior of migrated programs. Additionally, there may be utility in permitting “slightly wrong” type annotations. Finally, we would like to examine other deep learning models and type migration tasks beyond type annotation prediction.

References

- 1 Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *Principles of Programming Languages (POPL)*, 2011. doi:10.1145/1926385.1926437.
- 2 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005. doi:10.1007/11531142_19.

- 3 Luke Autry. How we failed, then succeeded, at migrating to TypeScript. <https://heap.io/blog/migrating-to-typescript>, 2019. Accessed: 2022-12-01.
- 4 Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient Training of Language Models to Fill in the Middle, 2022. doi:10.48550/arXiv.2207.14255.
- 5 Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. SantaCoder: don't reach for the stars!, 2023. doi:10.48550/arXiv.2301.03988.
- 6 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014. doi:10.1007/978-3-662-44202-9_11.
- 7 Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical Optional Types for Clojure. In *European Symposium on Programming (ESOP)*, 2016. doi:10.1007/978-3-662-49498-1_4.
- 8 Ryan Burgess, Joe King, Stacy London, Sumana Mohan, and Jem Young. TypeScript migration - Strict type of cocktails. <https://frontendhappyhour.com/episodes/typescript-migration-strict-type-of-cocktails>, 2022. Accessed: 2022-12-01.
- 9 John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating Gradual Types. *Proc. ACM Program. Lang.*, 2(POPL), 2018. doi:10.1145/3158103.
- 10 Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI)*, 1991. doi:10.1145/113445.113469.
- 11 Mauricio Cassola, Agustín Talagorria, Alberto Pardo, and Marcos Viera. A Gradual Type System for Elixir. In *Brazilian Symposium on Context-Oriented Programming and Advanced Modularity (SBLP)*, 2020. doi:10.1145/3427081.3427084.
- 12 Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.*, 3(POPL), 2019. doi:10.1145/3290329.
- 13 Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. Type Inference for Static Compilation of JavaScript. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2016. doi:10.1145/2983990.2984017.
- 14 Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133872.
- 15 Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012. doi:10.1145/2384616.2384659.
- 16 Asger Feldthaus and Anders Møller. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2014. doi:10.1145/2660193.2660215.
- 17 Cormac Flanagan. *Effective Static Debugging via Componential Set-based Analysis*. PhD thesis, Rice University, 1997. URL: <https://users.soe.ucsc.edu/~cormac/papers/thesis.pdf>.
- 18 Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A Generative Model for Code Infilling and Synthesis. In *International Conference on Learning Representations (ICLR)*, 2023. doi:10.48550/arXiv.2204.05999.

- 19 Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009. doi:10.1145/1640089.1640110.
- 20 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. In *Symposium on Applied Computing (SAC)*, 2009. doi:10.1145/1529282.1529700.
- 21 Ronald Garcia and Matteo Cimini. Principal Type Schemes for Gradual Programs. In *Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2676726.2676992.
- 22 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming (ESOP)*, 2011. doi:10.1007/978-3-642-19718-5_14.
- 23 Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep Learning Type Inference. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2018. doi:10.1145/3236024.3236051.
- 24 Kevin Jesse, Premkumar Devanbu, and Anand Ashok Sawant. Learning To Predict User-Defined Types. *IEEE Transactions on Software Engineering (TSE)*, 2022. doi:10.1109/TSE.2022.3178945.
- 25 Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. Learning Type Annotation: Is Big Data Enough? In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2021. doi:10.1145/3468264.3473135.
- 26 Milod Kazerounian, Jeffrey S. Foster, and Bonan Min. SimTyper: Sound Type Inference for Ruby Using Type Equality Prediction. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021. doi:10.1145/3485483.
- 27 Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. Sound, Heuristic Type Annotation Inference for Ruby. In *Dynamic Languages Symposium (DLS)*, 2020. doi:10.1145/3426422.3426985.
- 28 Erik Krogh Kristensen and Anders Møller. Inference and Evolution of TypeScript Declaration Files. In *Fundamental Approaches to Software Engineering (FASE)*, 2017. doi:10.1007/978-3-662-54494-5_6.
- 29 Erik Krogh Kristensen and Anders Møller. Type Test Scripts for TypeScript Testing. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. doi:10.1145/3133914.
- 30 Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Dynamic Languages Symposium (DLS)*, 2013. doi:10.1145/2578856.2508170.
- 31 Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. Gradual Soundness: Lessons from Static Python. *The Art, Science, and Engineering of Programming*, 7(1), 2022. doi:10.22152/programming-journal.org/2023/7/2.
- 32 Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *International Conference on Software Engineering (ICSE)*, 2019. doi:10.1109/ICSE.2019.00045.
- 33 Meta Platforms, Inc. Pyre: A performant type-checker for Python 3. <https://pyre-check.org/>. Accessed: 2022-12-01.
- 34 Zeina Migeed and Jens Palsberg. What Is Decidable about Gradual Types? *Proc. ACM Program. Lang.*, 4(POPL), 2020. doi:10.1145/3371097.
- 35 Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic Type Inference for Gradual Hindley–Milner Typing. *Proc. ACM Program. Lang.*, 3(POPL), 2019. doi:10.1145/3290331.
- 36 Thomas Moore. How We Completed a (Partial) TypeScript Migration In Six Months. <https://blog.abacus.com/how-we-completed-a-partial-typescript-migration-in-six-months/>, 2019. Accessed: 2022-12-01.
- 37 Guilherme Ottoni. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *Programming Language Design and Implementation (PLDI)*, 2018. doi:10.1145/3192366.3192374.

37:28 Do Machine Learning Models Produce TypeScript Types That Type Check?

- 38 Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints, 2021. doi:10.48550/arXiv.2004.00348.
- 39 Mihai Parparita. The Road to TypeScript at Quip, Part Two. <https://quip.com/blog/the-road-to-typescript-at-quip-part-two>, 2020. Accessed: 2022-12-01.
- 40 Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. Solver-Based Gradual Type Migration. *Proc. ACM Program. Lang.*, 5(OOPSLA), 2021. doi:10.1145/3485488.
- 41 Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural Type Prediction with Search-Based Validation. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, 2020. doi:10.1145/3368089.3409715.
- 42 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The Ins and Outs of Gradual Type Inference. In *Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103656.2103714.
- 43 Felix Rieseberg. TypeScript at Slack. <https://slack.engineering/typescript-at-slack/>, 2017. Accessed: 2022-12-01.
- 44 Sergii Rudenko. ts-migrate: A Tool for Migrating to TypeScript at Scale. <https://medium.com/airbnb-engineering/ts-migrate-a-tool-for-migrating-to-typescript-at-scale-cd23bfeb5cc>, 2020. Accessed: 2022-12-01.
- 45 Claudiu Saftoiu. JSTrace: Run-time Type Discovery for JavaScript. Master’s thesis, Brown University, 2010. URL: <https://cs.brown.edu/research/pubs/theses/ugrad/2010/saftoiu.pdf>.
- 46 Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, 2006. URL: <http://schemeworkshop.org/2006/13-siek.pdf>.
- 47 Jeremy G. Siek and Manish Vachharajani. Gradual Typing with Unification-Based Inference. In *Dynamic Languages Symposium (DLS)*, 2008. doi:10.1145/1408681.1408688.
- 48 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Principles of Programming Languages (POPL)*, 2008. doi:10.1145/1328438.1328486.
- 49 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Trust, but Verify: Two-Phase Typing for Dynamic Languages. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.52.
- 50 Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*, 2020. doi:10.48550/arXiv.2005.02161.
- 51 Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed Messages: Measuring Conformance and Non-Interference in TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2017. doi:10.4230/LIPIcs.ECOOP.2017.28.
- 52 Jake Zimmerman. Sorbet: Stripe’s type checker for Ruby. <https://stripe.com/blog/sorbet-stripes-type-checker-for-ruby>, 2022. Accessed: 2022-12-01.

Building Code Transpilers for Domain-Specific Languages Using Program Synthesis

Sahil Bhatia ✉

University of California, Berkeley, CA, USA

Sumer Kohli ✉

University of California, Berkeley, CA, USA

Sanjit A. Seshia ✉

University of California, Berkeley, CA, USA

Alvin Cheung ✉

University of California, Berkeley, CA, USA

Abstract

Domain-specific languages (DSLs) are prevalent across many application domains. Such languages let developers easily express computations using high-level abstractions that result in performant implementations. To leverage DSLs, however, application developers need to master the DSL's syntax and manually rewrite existing code. Compilers can aid in this effort, but part of building a compiler requires transpiling code from the source code to the target DSL. Such transpilation is typically done via pattern-matching rules on the source code. Sadly, developing such rules is often a painstaking and error-prone process.

In this paper, we describe our experience in using program synthesis to build code transpilers. To do so, we developed METALIFT, a new framework for building transpilers that transform general-purpose code into DSLs using program synthesis. To use METALIFT, transpiler developers first define the target DSL's semantics using METALIFT's specification language, and specify the search space for each input code fragment to be transpiled using METALIFT's API. METALIFT then leverages program synthesizers and theorem provers to automatically find transpilation expressed in the target DSL that is provably semantic equivalent to the input code. We have used METALIFT to build three DSL transpilers targeting different programming models and application domains. Our results show that the METALIFT-based compilers can translate many benchmarks used in prior work created by specialized implementations, but can be built using orders-of-magnitude fewer lines of code as compared to prior work.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Program Synthesis, Code Transpilation, DSLs, Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.38

Category Experience Paper

Supplementary Material *Software*: <https://github.com/metalift/metalift>

Funding This work was supported in part by a Google BAIR Commons project, by the DARPA LOGiCS project, by the NSF FMitF-1837132, IIS-1955488, IIS-2027575, CCF-1723352, ARO W911NF2110339, ONR N00014-21-1-2724, and DOE award DE-SC0016260.

Acknowledgements We would like to thank ShadaJ Laddad, Maaz Ahmad and anonymous reviewers for their insightful feedback.

1 Introduction

Domain-specific languages (DSLs) are now popular means to develop applications across many domains. Besides improving programmability, modern DSLs often expose *domain-specific optimizations* via their interfaces for applications to leverage specialized hardware accelerators [16, 32, 6, 21], or domain-specific code transformation [15, 43, 12].



© Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 38; pp. 38:1–38:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Yet, to capitalize on the benefits provided by DSLs, developers must learn the interfaces provided by the target DSL. For existing applications, developers need to painfully reverse-engineer legacy code, potentially convoluted with optimizations, before rewriting it, only to realize that newly emerging frameworks can turn freshly rewritten code into legacy applications. Needless to say, each rewrite is another opportunity to introduce bugs into the application.

The classical mechanism to alleviate this problem is for DSL designers to build pattern-driven transpilers that translate programs, say written in general-purpose languages, into their DSLs [4]. While such transpilers are essentially part of all modern optimizing compilers, they often require developing a complex network of inter-connected translation rules [22, 31, 39], which is a highly tedious and error-prone task.

Instead of transpilation rules, researchers have leveraged advances in program synthesis [11, 17] for code transpilation, where the idea is to replace the rule-matching machinery with a code synthesizer that finds programs written in the target language that are semantically equivalent to the input code fragment [23, 14, 27, 20, 3]. However, using such techniques involves encoding the semantics of the input program as a synthesis problem. Furthermore, building such transpilers requires implementing specialized synthesis procedures, which relies on specialized knowledge of synthesis algorithms that most transpiler designers do not possess.

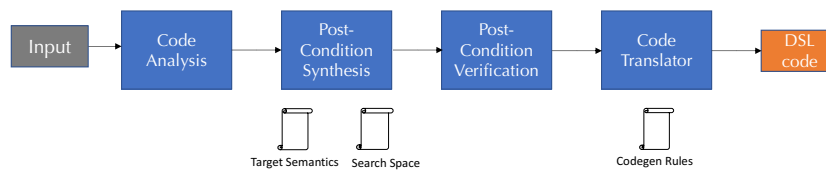
In this paper, we describe our experience in building code transpilers using program synthesis. To do so in a systematic manner, we designed a new framework called METALIFT. Our goal in building METALIFT is to free transpiler developers from designing a myriad of pattern matching rules for transpilation, while making synthesis technology easily accessible for code translation. Given input code written in the source language (METALIFT currently supports LLVM), developers can use METALIFT to implement a code transpiler that uses program synthesis to search for code in the target language that is *provably semantically equivalent* to the input. To use METALIFT, developers first define the semantics of their target DSL using MTL, i.e., METALIFT’s Specification Language. Then, using METALIFT’s search space API, developers specify the search space of DSL programs for each input. The search space can be constructed programmatically by analyzing each input code to transpile. Given the DSL definition and search space description, developers then write a transpiler driver that orchestrates the transpilation process.

METALIFT is designed with developer usability in mind in constructing transpilers. As we will discuss in detail, MTL is designed as a specification language embedded within Python for ease of use. As METALIFT focuses on ensuring semantic equivalency between the source and transpiled code, MTL consists of a small number of constructs, and is designed to be high-level so that developers can easily use it to express the semantics of each construct in their target DSL, and yet simple enough for METALIFT to compile down as the input to different synthesizers and verifiers. Likewise, the METALIFT’s API is also designed to abstract away the details of synthesis and verification from the developer.

To evaluate METALIFT, we have used the framework to reproduce three different transpilers described in prior work spanning multiple application domains and programming models. Our evaluation shows that they require order-of-magnitude fewer lines of code to build as compared to prior work, with the resulting transpiler generating the same (or very similar) code as the original implementations.

In summary, this paper makes the following contributions:

- We design MTL for developers to specify the semantics of different constructs in their DSLs. The design of MTL is general enough to support many real-world DSLs, yet simple enough to be translated as the input to various program synthesizers and verification engines.



■ **Figure 1** An overview of the METALIFT architecture.

- We describe METALIFT, a unified framework for developing transpilers for DSLs. Rather than designing pattern matching rules, METALIFT enables designers to use program synthesis to easily translate input code to their DSLs without requiring any program synthesis expertise. Furthermore, the translations generated by METALIFT are formally verified for semantic equivalence to the source code, so developers are assured of an accurate translation.
- We show how METALIFT generalizes previous work on building transpilers using program synthesis by creating two DSL transpilers that translate from general-purpose code to these DSLs. These three DSLs are aimed at different application domains. Our evaluations demonstrate how METALIFT dramatically reduces the effort required to build these compilers.

We organize the rest of the paper as follows. We provide an overview of METALIFT (Section 2). We describe METALIFT in more detail (Section 3) using a representative example. We evaluate METALIFT in Section 4 and review the prior approaches for building transpilers in Section 5. Finally, we conclude (Section 6) with directions for future work.

2 Overview

In this section, we provide an overview of the METALIFT framework. The high-level architecture of METALIFT is shown in Figure 1. For the majority of the paper, we use the example in Figure 2 as our running example and describe how to build a transpiler using METALIFT that translates sequential Java code to the Spark DSL. Spark [43] provides users with an interface to efficiently process large-scale distributed computations in a parallel and fault-tolerant manner.

Concretely, the example in Figure 2a takes as input a list of words and counts the frequency of each word in input list. Figure 2b shows the equivalent implementation using map reduce operators in the Spark DSL. The `map` operator returns `(word, 1)` for each word in the input list, and the `reduceByKey` operator then uses the reducer function `(v1 + v2)` to aggregate each unique key in the map operator’s output.

The Mold compiler [31] has implemented a syntax-driven compiler that automatically translates sequential Java code to Spark. To perform this translation, Mold uses rewrite rules that pattern match on the input source code. However, these rules can be hard to implement as

- 1) they must be expressive in order to capture all of the different coding patterns,
- 2) they must ensure semantic equivalence to the source code, and
- 3) they must be maintained as the DSLs change.

For instance, as described in their paper, Mold requires 22 different rules to generate the corresponding Spark program for the same word count program shown in Figure 2a.

38:4 Building Code Transpilers for Domain-Specific Languages Using Program Synthesis

```
1 map<string, int> countWords(vector<string> words) {
2     map<string, int> counts;
3     for (int j = 0; j < words.size(); j++) {
4         string word = words[j];
5         int prev = 0;
6         if (counts.find(word) != counts.end())
7             prev = counts[word];
8         counts[word] = prev + 1;
9     }
10    return counts;
11 }
```

(a) Input Source Code: Sequential C++ Code (Java benchmarks are converted to C++ since the METALIFT front-end currently supports C++).

```
1 Map<String, Integer> countWords(List<String> words) {
2     Map<String, Integer> counts = new HashMap<String, Integer>();
3     counts = words.mapToPair(v -> new Tuple2<String,Integer>(v, 1))
4         .reduceByKey((v1,v2) -> v2 + v1).collectAsMap();
5     return counts;
6 }
```

(b) Output: Apache Spark Code.

■ **Figure 2** Translation from sequential C++ code to Apache Spark DSL.

```
1 # target DSL definition, see Figure 11
2 # grammar description, see Figure 13
3 # code generation rules, see Figure 18
4
5 def transpiler(source): # driver program
6     liveVars, modVars, VC = analyze(source)
7     verifiedSummaries = synthesize(VC, targetLang(), grammar(liveVars, modVars))
8     transpiledCode = codeGen(verifiedSummaries)
9     return transpiledCode
```

■ **Figure 3** Example of the METALIFT driver code to transpile the running example shown in Figure 2.

Instead of designing syntax-driven rules, developers can use METALIFT to build this transpiler. METALIFT leverages program synthesis to search for transformations that are semantically equivalent to the source code. Figure 3 shows the driver code developers will write to implement a compiler using METALIFT, where they provide the following inputs to METALIFT:

1. **Target DSL Definition.** First, using MTL and the programming interface provided by METALIFT, developers define the semantics of the operators in their target DSL. Each operator represents a program construct in the target language. For instance, for our Spark compiler, we define the semantics of the map and reduce operators as shown in Figure 11. In Section 3.3, we discuss how MTL can be used to define the semantics for each construct in the target language.
2. **Search Space Definition.** Besides the target language, developers also define the search space description to guide METALIFT's synthesis engine (Figure 13). The search space provides the space of possible programs in which the synthesis engine can look for

equivalent program transformations. Using the variable information returned during the analysis phase performed by METALIFT, developers can use MTL to describe the search space. This will be discussed further in Section 3.6.

3. **Code Generation Rules.** The final input that developers provide is the syntax-driven rules to translate the summaries synthesized by METALIFT into executable target DSL code. Note that translating from summaries is much easier than translating directly from source code as the summaries are already encoded using the target DSL’s operators. Section 3.9 provides more details about how these rules can be implemented.

Developers then write a transpiler driver that invokes the inputs mentioned above (as shown in Figure 3), using METALIFT’s API as follows:

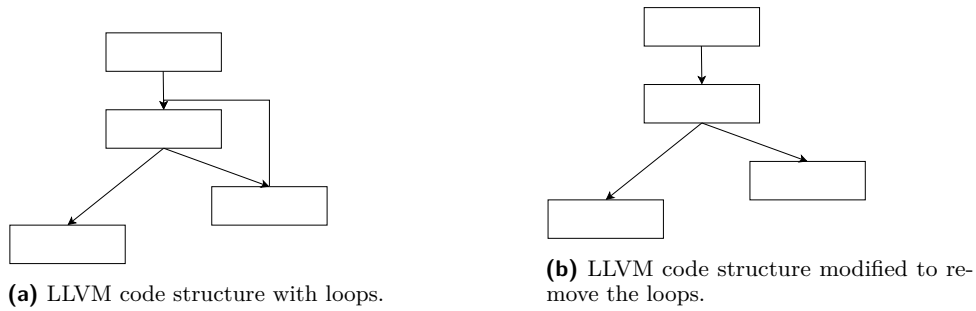
1. **Code Analysis.** METALIFT’s Analysis API takes as input the source code to be translated and compiles it into LLVM intermediate representation (IR), as shown in Line 6 in Figure 3. We use LLVM IR because it allows for the compilation of multiple general-purpose languages (e.g., C, C++, Fortran) using various front ends, giving METALIFT the flexibility to support multiple languages on the source side. The goal of the analysis phase is to compute verification conditions from the LLVM IR. Verification conditions (VCs) are logical statements that assert that the program is correct with respect to the given pre-condition and post-condition if they hold. The VCs serve as the specification for the synthesizer. The analysis phase also returns information about the output variables and the variables being modified in the source code. We describe METALIFT’s analysis phase in more detail in Section 3.1.
2. **Synthesis and Verification.** Once the analysis phase returns the VCs, developers then invoke METALIFT’s synthesis API, as shown in Line 7 in Figure 3. The VCs generated during the analysis phase form the synthesizer’s specification. METALIFT then synthesizes program summaries that meet these specifications. To make the search tractable, METALIFT’s synthesizer uses the search space description provided by the developers. These summaries are restricted to be expressed using only the operators in the target DSL defined by the developers. Logically, a program summary is the post-condition that captures the program’s final states after execution. Finally, METALIFT formally verifies that the generated program summary is semantically equivalent to the input source code using an automated theorem prover. Synthesis and verification phases of METALIFT are discussed in Section 3.7 and Section 3.8, respectively.
3. **Code Generator.** In the final step as shown in Line 8 in Figure 3, METALIFT invokes the user-defined code generation rules to convert the verified summaries into executable DSL code. The compiled code is then returned back to the user.

3 Framework

In this section, we discuss each component of METALIFT in detail by building a transpiler for our example in Figure 2.

3.1 Analysis

The front-end of METALIFT takes as input the source code written in a general-purpose languages which the developers want to transpile to their DSLs. As mentioned, our current prototype supports languages that can be compiled to LLVM IR. Once compiled to the LLVM IR, the next step in the analysis phase is to augment the generated LLVM IR to enable computation of verification conditions.



■ **Figure 4** Transformations performed on loop constructs during analysis phase.

Prior work [18, 24] has recently introduced frameworks for automated verification of software systems. These frameworks takes as input the description of the system to be verified, the specification that the system must satisfy, and internally reduces the problem to symbolic constraints that are then discharged to a theorem prover for verification. However, METALIFT cannot reuse any of these front-ends for generating VCs as they rely on the source code already having annotated with invariants and post-conditions. METALIFT instead models invariants and post-conditions as predicates that take in all live variables at the point when they are declared, and leave the body of the predicates to be *synthesized* later on, to be explained in Section 3.7.

Verification Conditions. In Floyd-Hoare logic (FHL) [19], the verification problem is abbreviated using the Hoare Triple $\{A\} S \{P\}$. To establish the validity of a Hoare triple, we need to prove that for all executions starting from states that satisfy A (pre-condition), after executing statement S , should satisfy P (post-condition). An example of a valid Hoare triple is $\{y \leq x\} z := x; z := z + 1 \{y < z\}$. This problem can be further simplified as finding a Boolean predicate which characterizes the set of pre-conditions from which every execution of S would lead a to state that satisfies P . These Boolean predicates are known as verification conditions. Formally, we can represent this as proving the following logical statement as $A \rightarrow VC(S, P)$.

An additional predicate called loop invariant is required for programs with loop constructs to prove that the post-condition is valid regardless of the number of iterations of the loop. Thanks to the efficient theorem provers [42, 7], such inference rules provided by FHL can be encoded in solvers such that any Hoare triple can be mechanically checked for its validity.

In METALIFT, S corresponds to the program statements in the input code to be transpiled, while A and P correspond to expressions written using MTL to be discussed in Section 3.3. Our goal is to synthesize a post-condition for each input code that are expressed using the target language constructs provided by the user, while ensuring that it forms a valid Hoare triple together with the input code S and pre-condition A .

Transformations for Loop Constructs. For programs that do not have any loop constructs, generating VCs is straightforward. However, programs with looping constructs requires more processing. In Figure 4a, we show a simplified control-flow graph of LLVM basic blocks for a program with loops. For such programs, we first identify the back edges, *i.e.*, the edge from the loop body to the loop head, and remove that edge to transform the bitcode into an acyclic graph. To preserve the semantics of the loop after removing the back edge, we annotate the start of the loop head block with *havoc* statement as shown in Figure 4b. The

havoc statement is introduced for all variables modified in the loop to update their contents to fresh symbolic variables. Doing so allow us to mimic the effect of executing an arbitrary iteration of the loop, which will be crucial in generating the correct verification condition.

The acyclic graph now represents any arbitrary iteration of the loop. To prove the validity of the Hoare triple $\{A\} \text{ while } G \text{ do } S \{P\}$, we must identify a *loop invariant* that holds at the beginning of the loop, at every iteration of the loop, and at the end of the loop terminates. As the input code is not annotated with loop variants, to generate VCs which can prove the validity of the described Hoare triple, we add the following annotations (shown in **bold** in Figure 6) to the LLVM code to be transpiled:

1. assert that invariant holds before the entry to the loop (Line 9)
2. havoc the variables being modified inside the loop (Line 13)
3. assume invariant holds before the execution the body (Line 15) and assume loop guard is true (Line 23)
4. assert invariant holds after the execution of the loop body (Line 35)
5. assert invariant holds after the loop exits (Line 38)
6. assume loop guard is false and assert that the post-condition holds before the program exits (Line 40)

3.2 Verification Condition Generation

There are several ways to prove functional equivalence of the source and target program. We adopt VCs as the specification for checking the functional equivalence. VCs enable us to prove complete functional equivalence between the source and generated target code, meaning that we can generate a proof of equivalence for all possible inputs. While there are other potential means to provide specifications, such as using the inputs and outputs from test cases or relying on the equivalence between a general-purpose program and its corresponding DSL program, these approaches have serious drawbacks:

1. Using testing as the specification only guarantees correctness for a finite set of inputs since it's not feasible to generate all possible test cases. Automating test case generation is also not always reliable, as it may not cover all paths of the program, and running these programs might not always be feasible. As a result, the synthesized program will only be semantically equivalent modulo the inputs and outputs from the test cases that were used as specification.
2. Checking equivalence between the source and target programs directly would require encoding the semantics of different constructs appearing in the source, and the most popular symbolic synthesizers [37, 41] or verifiers [42, 7] do not support semantic reasoning of loops. These symbolic synthesizers only reason about loops after they have been unrolled for a finite amount of iterations, effectively converting the loop into straight-line code. Because of this, it is challenging to check equivalence for programs with loops without generating VCs and loop invariants, as doing so only provides guarantees up to a certain bound.

Our VC generation algorithm is inspired from [8]. The LLVM compilation process generates LLVM bitcode which is represented using the LLVM IR. Figure 6 shows the abridged LLVM bitcode for the source code in Figure 2a. LLVM bitcode contains multiple basic blocks, i.e, contiguous sequence of LLVM IR instructions with just one entry and one exit point. A basic block after the transformation in the analysis phase has the general structure shown in Figure 5a.

```

1 blk: bbid
2   assume ea;
3   %i = ...;
4   %i1 = ...;
5   assert eb
6   br label %bbid'

```

(a) Basic Block structure after the analysis phase.

```

1 blk: bbid
2   assume ea;
3   assume %i = ...;
4   assume %i1 = ...;
5   assert eb
6   br label %bbid'

```

(b) Basic Block structure after converting instructions to assumes.

■ **Figure 5** Basic block structure of the LLVM bitcode.

```

1 blk: bb_start
2   %i = alloca %list* ;i = input variable words
3   %i1 = alloca %dict* ;i1 = output variable counts
4   %i2 = alloca i32 ;i2 = loop counter j
5   store %list* %arg, %list** %i
6   store call %dict* newMap(), %dict** %i1
7   store i32 0, i32* %i2
8   ;invariant is true before the execution of the loop
9   (assert call inv (load i1) (load i2) arg)
10  br label %bb_head
11
12 blk: bb_head
13  (havoc i1 i2)
14  ;invariant is true at the start of the loop body
15  (assume call inv ((load i1) (load i2) arg))
16  %i7 = load i32, i32* %i2
17  %i8 = load %list*, %list** %i
18  %i9 = call i32 @length(%list* %i8)
19  %i10 = icmp slt i32 %i7, %i9 ;j < words.size()
20  br i1 %i10, label %bb_body1, label %bb_exit
21
22 blk: bb_body1
23  (assume i10) ;loop guard is true
24  ;instructions to update the counts in the output map (not shown for brevity)
25  %i11 = ...
26  %i12 = ...
27  br label %bb_body2
28
29 ;instructions to update the counts in the output map
30 blk: bb_body2
31  %i28 = load i32, i32* %i2, align 4
32  %i29 = add nsw i32 %i28, 1
33  store i32 %i29, i32* %i2, align 4
34  ;invariant is true after executing the body of the loop
35  (assert call inv ((load i1) (load i2) arg))
36
37 blk: bb_exit
38  (assume not(i10)) ;loop guard is false
39  %i31 = load %dict*, %dict** %i1
40  (assert call ps (i31 arg)) ;post-condition is true

```

■ **Figure 6** LLVM bitcode for the source code in Figure 2a. Bitcode is annotated with assumes, asserts and calls to inv and ps for generating the verification conditions.

Our VC generation algorithm computes an assertion for each of the basic blocks. For each basic block BB in our bitcode we introduce a new Boolean variable BB_{ok} and the VC for that block is expressed as:

$$BB_{ok} = VC(S, \bigwedge_{B \in Succ(A)} B_{ok}) \quad (1)$$

where S represents all the instructions of the block and $Succ(A)$ represents the set of successor basic blocks of block BB .

In addition to the transformations during the analysis phase, we convert all the instructions in the block to assume statements, as illustrated in Figure 5b.

We then symbolically execute all the instructions in the block to generate the VCs. For symbolic execution, we maintain two dictionaries to model the *memory* (m) and *registers* (r). m maps memory cells to their values, while r keeps track of the results of the instructions' execution. The state of the symbolic executor is maintained as a quadruple $\langle m, r, a, b \rangle$ that represents the state of the memory, registers, assumptions encountered, and assertions encountered thus far during symbolic execution, respectively. a then represents the final VC for the input code fragment after symbolic execution terminates.

$$\begin{array}{c}
\text{STORE} \\
\frac{m' = m[r[i] \mapsto v]}{\llbracket \text{store } v \ i \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m', r, a, b \rangle} \\
\\
\text{MEMORY ALLOCATION} \\
\frac{\text{fresh } l \quad r' = r[i \mapsto l] \quad m' = m[l \mapsto \perp]}{\llbracket i = \text{alloca } t \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m', r', a, b \rangle} \\
\\
\text{LOAD} \\
\frac{r' = r[i \mapsto m[i_a]]}{\llbracket i = \text{load } i_a \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m, r', a \wedge (i = m[i_a]), b \rangle} \\
\\
\text{HAVOC} \\
\frac{\text{fresh } v' \quad m' = m[v \mapsto v']}{\llbracket \text{havoc } v \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m', r, a, b \rangle} \\
\\
\text{ARITHMETIC OPERATORS (AOP)} \\
\frac{v = \text{aop}(r[i_a], r[i_b]) \quad r' = r[i \mapsto v]}{\llbracket i = \text{aop } i_a \ i_b \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m, r', a, b \rangle} \\
\\
\text{BINARY COMPARISONS (BOP)} \\
\frac{v = \text{bop}(r[i_a], r[i_b]) \quad r' = r[i \mapsto v]}{\llbracket \text{icmp } \text{bop } i_a \ i_b \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m, r', a \wedge (i = v), b \rangle} \\
\\
\text{ASSUME} \\
\frac{}{\llbracket \text{assume } e \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m, r, a \wedge \llbracket e \rrbracket, b \rangle} \\
\\
\text{ASSERT} \\
\frac{}{\llbracket \text{assert } e \rrbracket (\langle m, r, a, b \rangle) \Rightarrow \langle m, r, a, a \rightarrow b \wedge \llbracket e \rrbracket \rangle}
\end{array}$$

■ **Figure 7** Computing the VC via symbolic execution on LLVM instructions. The symbol \wedge and \rightarrow denotes logical And and Implies operator respectively.

In Figure 7, we describe how we compute the VC for each type of LLVM opcode, while Table 1 shows our symbolic executor in action using these rules. To illustrate, the STORE rule in Figure 7 states that if the current state of the symbolic executor is $\langle m, r, a, b \rangle$, to symbolically execute a `store` instruction that stores value v to the cell location stored at register i , we create a new memory m' where the cell $r[i]$ is mapped to the value v , with all other cells and their mappings remain unchanged from m . Similarly, the MEMORY ALLOCATION rule states that given the current symbolic executor state $\langle m, r, a, b \rangle$, to symbolically execute an `alloca` instruction that allocates new memory pointing to a value of type t at cell l with l to be stored in register i , we create a fresh symbolic cell l representing the pointer to the newly allocated memory, update the register state r to map register i to l instead, and update the memory state m where cell l points to uninitialized value \perp . The new symbolic state $\langle m', r', a, b \rangle$ is then returned.

As a concrete example, we show the state of the memory, registers and assumes expression after the execution of each instruction in the block `bb_head` in Figure 6. For this block, `asserts = null` and `Succ(A) = {bb_body1, bb_exit}`. The computed VC according to the Equation (1) and rules in Figure 7 would then be:

$$\begin{aligned} BB_head_{ok} &= (assumes \rightarrow bb_body1 \wedge bb_exit) \\ &= inv(i1_0, i2_1, arg) \wedge (i7 = i2_1) \wedge (i8 = arg) \wedge (i9 = length(arg)) \\ &\quad \wedge (i10 = i2_1 < length(arg)) \end{aligned} \quad (2)$$

■ **Table 1** Symbolic Execution for the block `BB_head` in Figure 6 using the inference rules defined in Figure 7. Each row depicts an LLVM instruction as well as the resulting *memory* (m) and *registers* (r) state after the instruction is symbolically executed. Due to space constraints, we do not show `inv` (and `True`) in all the execution steps.

havoc i1 i2 $m = [i1 \mapsto i1_0, i \mapsto arg, i2 \mapsto i2_1]$ $r = []$ <code>assumes = True</code>
assume inv ((load i1) (load i2) arg) $m = [i1 \mapsto i1_0, i2 \mapsto i2_1, i \mapsto arg]$ $r = []$ <code>assumes = True \wedge inv(i1_0, i2_1, arg)</code>
assume %i7 = load i32, i32* %i2 $m = [i1 \mapsto i1_0, i2 \mapsto i2_1, i \mapsto arg]$ $r = [i7 \mapsto i2_1]$ <code>assumes = True \wedge (i7 = i2_1)</code>
assume %i8 = load %list*, %list** %i $m = [i1 \mapsto i1_0, i2 \mapsto i2_1, i \mapsto arg]$ $r = [i7 \mapsto i2_1, i8 \mapsto arg]$ <code>assumes = (i7 = i2_1) \wedge (i8 = arg)</code>
assume %i9 = call i32 @length(%list* %i8) $m = [i1 \mapsto i1_0, i2 \mapsto i2_1, i \mapsto arg]$ $r = [i7 \mapsto i2_1, i8 \mapsto arg, i9 \mapsto length(arg)]$ <code>assumes = (i7 = i2_1) \wedge (i8 = arg) \wedge (i9 = length(arg))</code>
assume %i10 = icmp slt i32 %i7, %i9 $m = [i1 \mapsto i1_0, i2 \mapsto i2_1, i \mapsto arg]$ $r = [i7 \mapsto i2_1, i8 \mapsto arg, i9 \mapsto length(arg), i10 \mapsto (i2_1 < length(arg))]$ <code>assumes = (i7 = i2_1) \wedge (i8 = arg) \wedge (i9 = length(arg)) \wedge (i10 = i2_1 < length(arg))</code>

Similarly, the VCs for other basic blocks can be constructed. Once the VCs have been generated for the all basic blocks, the VC for the entire program can be expressed as $R \rightarrow BB_start_{ok}$ where R is the conjunction of VCs for each block and BB_start_{ok} is the Boolean variable introduce for the first block in the LLVM bitcode.

Note that the calls to invariant and post-conditions are just placeholders and they are synthesized during the synthesis phase of METALIFT. At the end of the analysis phase METALIFT would generate the following verification conditions:

1. $\forall \sigma. Pre(\sigma) \rightarrow Inv(\sigma)$
2. $\forall \sigma, \sigma'. Inv(\sigma) \wedge Body(\sigma, \sigma') \rightarrow Inv(\sigma')$
3. $\forall \sigma. Inv(\sigma) \rightarrow Post(\sigma)$

For simplicity, we do not show the verification conditions using LLVM basic block structure. The verification conditions logically state that

- 1) invariant must hold before the loop
- 2) invariant must be inductive i.e. it should be true at every iteration of the loop and
- 3) invariant must assert the post-condition upon exiting the loop.

Internally, METALIFT represents the VCs using MTL which we describe in the next section.

3.3 MTL

We now describe MTL in detail. METALIFT provides developers with an API to use the constructs in the MTL to define the operators of their target DSL. This is in contrast to the prior standalone transpilers [3, 35, 20] where the semantics of the target DSL are embedded within the transpiler and are not reusable. In Section 3.5, we show how developers can use MTL to describe the semantics of the operators in their target DSL.

$$\begin{aligned}
 e \in \text{expr} &::= l \mid \text{var} \mid e_1 \text{ bop } e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \neg e \mid \\
 &\quad f(e_1, e_2, \dots, e_n) \mid f_u(e_1, e_2, \dots, e_n) \mid e_{\text{list}} \mid e_{\text{map}} \mid e_{\text{tup}} \\
 e_{\text{list}} \in \text{listExpr} &::= \text{empty} \mid \text{length}(e) \mid \text{get}(e, i) \mid \text{append}(e, i) \mid \\
 &\quad \text{prepend}(i, e) \mid \text{concat}(e_1, e_2) \mid \text{tail}(e, i) \mid \text{take}(e, i) \\
 e_{\text{map}} \in \text{mapExpr} &::= \text{empty} \mid \text{get}(e_{\text{map}}, i) \mid \text{insert}(e_{\text{map}}, e_1, e_2) \\
 e_{\text{tup}} \in \text{tupExpr} &::= \text{make}(e_1, e_2, \dots, e_n) \mid \text{get}(e_{\text{tup}}, i) \\
 l \in \text{literal} &::= \text{True} \mid \text{False} \mid \text{Integer Constant} \\
 \text{bop} \in \text{binaryOp} &::= \text{and} \mid \text{or} \mid \text{implies} \mid = \mid + \mid - \mid * \mid / \mid > \mid <
 \end{aligned}$$

■ **Figure 8** Grammar definition of MTL.

MTL is a strongly-typed functional language that consists of three dialects: one for METALIFT to represent VCs internally, one for users to define their target language operators, and another for users to describe the search space. Figure 8 shows the core grammar of MTL that is shared between the three dialects. Even though MTL is a small language, it is expressive enough that can be used to specify the semantics of real-world DSLs, as we will discuss in Section 4.

As shown in Figure 8, the core expressions can be literals, variables passed in as arguments to the function, conditional expressions, expressions combined using Boolean operators (MTL supports all arithmetic, logical and relational operators). MTL also supports operations over list, tuples, and associative maps. It also supports uninterpreted functions (represented as f_u in Figure 8). Uninterpreted functions have no definition; their defining characteristic is simply

that they are deterministic, *i.e.*, for the same input, they always return the same output. They are useful in modeling certain operators (for example, complex math functions that are used in both the source and target languages) for which we only care about determinism.

MTL also provides support for three native data structures: lists, tuples and associative maps, along with some common operations over these data structures. For lists, in addition to standard list operations such as length, append, and value retrieval, MTL supports list comprehension functions, *tail*(*lst*, *index*), which returns all the elements after the first index elements of the list, and *take*(*lst*, *index*), which returns the first index elements of the list. For associative maps and tuples, MTL provides functions for inserting and retrieving values. Developers can also combine these data structures to construct nested data structures such as a list of tuples, a list of lists and a list of maps.

In summary, the different dialects of MTL are used for the following purposes during program transpilation:

1. To represent the VCs generated during the analysis phase.
2. To specify the semantics of the target DSL.
3. To describe the search space for valid program transformations.

3.4 Expressing Verification Conditions in MTL

$$v \in vcExpr := assert(e) \mid assume(e) \mid havoc(var)$$

■ **Figure 9** Additional constructs in MTL for verification condition generation.

Verification conditions generated during the analysis phase are encoded using the constructs in the MTL. As described in the Section 3.1, VC generation requires some additional constructs. We show these constructs in Figure 9 as *vcExpr*. These constructs include **assume**, **assert**, and **havoc**, which takes core expressions as arguments. Their semantics are defined earlier in Figure 7. *vcExpr* includes these constructs in addition to all the core constructs described in Figure 8. These constructs are useful for annotating the LLVM bytecode with invariant and post-condition placeholders. These are utilized by METALIFT internally and are not accessible to the developers via MTL API.

3.5 Expressing Target DSLs in MTL

As discussed in Section 2, METALIFT builds transpilers by leveraging program synthesis. METALIFT searches for program summaries that are semantically equivalent to the source code. Program summaries capture all the changes to the outputs of the source code and are expressed using the operators in the target DSLs. Once the program summaries are synthesized developers can write simple syntax-driven rules to translate them to the concrete syntax of the target DSL.

In order to synthesize the program summaries, the synthesizer requires the semantics of the DSL operators. Developers can implement their domain-specific operators using MTL as defined in Section 3.3. In Figure 11, we describe how map and reduce functions from the Spark DSL can be defined using MTL.

In METALIFT, each construct in the target language is defined using a dialect of MTL. In Figure 10, we show the two constructs, function declaration (*fnDecl*) and axioms (*axiom*), that are added to the core language described in Figure 8 to form this dialect for defining

$$\begin{aligned}
 f \in \text{fnDecl} & := \text{name}(arg_1 : t_1, arg_2 : t_2, \dots, arg_n : t_n) : t_r \rightarrow e \\
 a \in \text{axiom} & := \text{name}(arg_1 : t_1, arg_2 : t_2, \dots, arg_n : t_n) \rightarrow e
 \end{aligned}$$

■ **Figure 10** Additional constructs in MTL for defining the operators in the target language.

the operators in the target language. MTL supports recursive and higher-order functions. The target language definition is a collection of function declarations and axioms, where the body of both constructs are defined using *expr*'s in the core MTL.

In addition to the definitions of the operators in the target DSL, developers can provide properties specific to the operators in the DSL. Some of the properties that developers can define for the map and reduce operations are shown in Figure 11. These properties are helpful during the verification phase of METALIFT. We give more details about how these properties are used in Section 3.8.

Modern DSLs contain hundreds of functions and therefore searching for program summaries directly in the DSL APIs is not feasible. To make the synthesis algorithm tractable, METALIFT searches for summaries only using the operators defined by the developers. The high-level nature of MTL enables the developers to succinctly define the operators and abstract out the details of these operators in the DSL API. For example, the map definition in Figure 11 can represent the different variations of the map functions (*map*, *flatMap*, *mapToPair*) available in the Spark DSL.

3.6 Describing Search Space for Synthesis

Developers additionally provide the search space description for the synthesizer. Developers provide this description for program summaries as well as any invariant or functions to be synthesized (for example, in Spark, users can ask the synthesizer to generate the bodies of the λ_m and λ_r functions). In METALIFT, the search space is encoded using a context-free grammar (CFG), with the expression that needs to be synthesized at the top level, and the production rules specifying the possible values that the expression can take.

We use a different dialect of MTL for users to describe the search space. Built on top of the core language and as shown in Figure 12, MTL provides one non-deterministic construct, *Choose*, which the developers can use to describe the search space for the synthesis phase. The search space description is used to guide the synthesis process. Semantically, *Choose* lets the synthesizer return any of expressions in its argument list. It can be recursively nested or evaluate to one of the core expressions.

The search space description impacts the synthesizer's performance. If the grammar is too expressive, the synthesizer may take a long time to synthesize the correct expressions; conversely, if the grammar is too restrictive, the synthesizer may fail to find the correct expressions that satisfy the specification. Prior work [20, 2, 3, 14] specialized the grammar descriptions for the target domains and embedded them in the tools. Unfortunately, developers had no way of controlling the expressiveness of these grammars. METALIFT instead allows developers to programmatically control the grammars and even tune grammars for each benchmark separately.

The *Choose* function in Figure 12 is the basic construct that developers can use to describe the search space. *Choose*(e_c, e_c) allows the users to specify the set of candidate values for a particular expression. The candidate values (e in Figure 12) are described using the constructs in MTL (Figure 8). The synthesizer then selects from this set of possible

```

1 def targetLang():
2   fnDecl("map", lst, λm) =
3     if length(lst) == 0 then empty
4     else concat(λm(get(lst, 0)), map(tail(lst, 1)))
5
6   fnDecl("reduce", lst, λr) =
7     if length(lst) == 0 then 0
8     else λr(get(lst, 0), reduce(tail(lst, 1)))
9
10  fnDecl("reduceByKey", lst, λr) =
11    reduceByKeyHelper(lst, getKeys(lst), {}, λr)
12
13  fnDecl("reduceByKeyHelper", lst, keys, outMap, λr) =
14    if length(lst) == 0 then outMap
15    else reduceByKeyHelper(lst, tail(keys, 1),
16      insert(outMap, get(keys, 0), reduce(getVals(outMap, get(keys, 0)), λr)), λr)
17
18  fnDecl(getKeys, lst): ...
19
20  fnDecl(getVals, lst): ...
21
22 # operator specific properties
23 axiom("distributiveMapLemma", lst1, lst2) =
24   map(concat(lst1, lst2), λm) = concat(map(lst1, λm), map(lst2, λm))
25
26 axiom("distributiveReduceLemma", lst1, lst2, key) =
27   get(reduceByKey(concat(lst1, lst2), λr), key) =
28     get(reduceByKey(lst1, λr), key) +
29     get(reduceByKey(lst2, λr), key)
30
31 axiom("inductiveMapLemma", lst, index) =
32   implies(and((index ≥ 0), (index < length(lst))),
33     map(tail(lst, index), λm) = concat(λm(get(lst, index)),
34       map(tail(lst, index + 1), λm)))
35
36 axiom("inductiveMapReduceLemma", lst, index) =
37   implies(and((index ≥ 0), (index < length(lst))),
38     reduce(map(take(lst, index + 1), λm), λr) =
39     λr(reduce(map(take(lst, index), λm), λr), λm(get(lst, index))))

```

■ **Figure 11** Semantics of the operators in Spark DSL defined using constructs in MTL.

$$e_c \in \text{chooseExpr} \quad := \quad \text{Choose}(e_c, e_c) \mid e$$

■ **Figure 12** Additional constructs in MTL for description of the search space.

candidates, an expression that meets the specification. In terms of the CFG, *Choose* describes the production rules, i.e., the expansion rules for a non-terminal in the grammar. These grammars can potentially be recursive in nature. We provide a “bound” parameter in our API to control the depth of these grammar. This provides additional flexibility to the developers to programmatically define the depth of unrolling of their search space.

The VCs mentioned in Section 3.1 can be trivially satisfied by setting the invariant and post-condition to be True. The search space description helps prevent the synthesizer from generating such trivial solutions. At a high-level, METALIFT requires the post-condition search space to have the following structure:

$$\forall v_o \in \text{outputVars}. v_o = e_c \in \text{chooseExpr}$$

where *outputVars* is the set of all output variables in the source code. *e* in *chooseExpr* are expressions described using the core constructs in MTL using set of all the *input arguments* and the variables modified in the program. For the example in Figure 2, *outputVars* = {*count*} and *modVars* = {*j*, *word*}. Logically, it states that the output variables in the program should be expressed as an expression over the operators in the target DSL. METALIFT uses standard static analysis techniques to infer these variables during the analysis phase. In addition to restricting the synthesizer from generating trivial solutions, the search space description helps in scaling the synthesis problem.

Figure 13 shows one possible grammar description for the post-condition, invariant, mapper, and the reducer function for the translation problem described in Figure 2. The grammar for the program summary asserts that the output variable equals some MapReduce expression over the input data. The grammar for the invariant reasons about the bounds of the loop counter and how the count variable is modified in each iteration of the loop. The mapper function can return an empty list or a list of key-value pair and the reducer function can choose to reduce the input values using one of the arithmetic operators. In this grammar we have restricted the output to be a map, reduce or map followed by a reduce operation.

As mentioned earlier, developers can programmatically control the grammar structure. In the Spark transpiler, a few possible search strategies include incrementally increasing the number of map reduce operations that can be used to express the output variable or incrementally increasing the number of emit statements that the mapper function can use. Figure 14 shows how the driver code in Figure 3 can be modified (less than 20 LOC) to implement the incremental search for number of emit statements. In Figure 14, each iteration of loop increments the number of emits by 1 until the synthesizer finds the semantically equivalent program summaries. This programmatic definition of grammar allows developers to experiment with the synthesis engine without needing to be synthesis experts. Some of these strategies were encoded in the previous tools [3] to make the search tractable.

METALIFT automatically generates the program summaries once the developers have provided the semantics of their operators and the search space description.

3.7 Synthesis

We now discuss the synthesis phase of METALIFT. A typical program synthesis problem is characterized by three parameters: the specification, space of possible programs and the search techniques used by the synthesizer to search for candidate solutions. In METALIFT, the specification are the VCs generated from the source code during the analysis phase and the space of possible programs is represented by the operators and search space specified by the developer. During the synthesis phase, METALIFT uses this information to synthesize the program summaries and any necessary invariants. Essentially, program summaries are logical statements asserting what should be true if the program terminates and it should hold for all possible executions of the program starting from a state that satisfies the pre-condition.

Formally, the synthesis problem can be stated as

$$\exists ps, inv_1, inv_2, \dots, inv_n. \forall \sigma. VC(S, ps, inv_1, inv_2, \dots, inv_n, \sigma) \quad (3)$$

The goal of the synthesizer is to infer the definitions of *ps* and *inv* (in case of programs with loops) such that for all program states σ , the verification conditions (generated during the analysis phase) for a given input source code *S* is true.

38:16 Building Code Transpilers for Domain-Specific Languages Using Program Synthesis

```
1 def grammar(j, counts, words):
2   def psGrammar(counts, words):
3     MR = Choose(map(words,  $\lambda_m$ ), reduce(words,  $\lambda_r$ ), reduceByKey(map(words,  $\lambda_m$ ),  $\lambda_r$ ))
4     ps = Choose(counts = MR)
5     return ps
6
7   def invGrammar(j, counts, words):
8     intConst = Choose(0,1)
9     listChoice = Choose(words, take(words,j), tail(words,j))
10    counterExp1 = Choose((j  $\leq$  intConst), (j  $\geq$  intConst),
11                        (j < intConst), (j > intConst))
12    counterExp2 = Choose((j  $\leq$  length(words)), (j  $\geq$  length(words)),
13                        (j < length(words)), (j > length(words)))
14    outExp = Choose(counts = MR)
15    MR = Choose(map(listChoice,  $\lambda_m$ ), reduce(listChoice,  $\lambda_r$ ),
16                reduceByKey(map(listChoice,  $\lambda_m$ ),  $\lambda_r$ ))
17    inv = Choose(And(counterExp1, counterExp2, outExp))
18    return inv
19
20  def mapperGrammar(v):
21    litChoice = Choose(0,1,v)
22    Emit = Choose(Tuple(litChoice, litChoice), litChoice)
23     $\lambda_m$  = Choose(append(empty, Emit))
24    return  $\lambda_m$ 
25
26  def reducerGrammar(v1, v2):
27     $\lambda_r$  = Choose((v1 + v2), (v1 - v2), (v1 * v2), (v1 / v2))
28    return  $\lambda_r$ 
```

■ **Figure 13** Search space description for the example in Figure 2 using MTL.

```
1 def searchSpace(liveVars, modVars, numEmits):
2   def generateEmits(numEmits):
3     if numEmits == 1: return append([], Emit)
4     else:
5       return append(Emit, generateEmits(numEmits-1))
6
7   def mapperGrammar(v, numEmits):
8     litChoice = Choose(0,1,v)
9     Emit = Choose(Tuple(litChoice, litChoice), litChoice)
10     $\lambda_m$  = Choose(generateEmits(numEmits))
11    return  $\lambda_m$ 
12
13  def transpiler(source): # incremental search driver program
14    liveVars, modVars, VC = analyze(source)
15    numEmits = 1
16    isSynthesized = False
17    while(isSynthesized == False):
18      grammar = searchSpace(liveVars, modVars, numEmits)
19      verifiedSummaries = synthesize(VC, targetLang(), grammar)
20      if verifiedSummaries != None:
21        isSynthesized = True
22      else:
23        numEmits += 1
24    transpiledCode = codeGen(verifiedSummaries)
25    return transpiledCode
```

■ **Figure 14** Modified driver code to implement incremental grammar search for controlling the number of emits in the λ_m function.

Many search techniques have been proposed to solve the problem stated in Equation (3). These include enumerative search [5], deductive search [30], constraint solving [38], statistical [34] and neural approaches [26]. Currently, METALIFT relies on Rosette, an off-the shelf synthesis solver [41] to perform this search; in principle, it could use any synthesis solver supporting the needed theories. To leverage Rosette as the synthesizer, we need to translate the MTL descriptions in the Rosette language. Due to the highly-syntactic nature of the Rosette language, this translation is straightforward. Synthesizer enumerates candidates for ps (and $invs$) from the search space described by the developer until it finds one that satisfies the specification.

Currently available synthesizers have limited support for handling recursive functions in the search space. Internally, synthesizers use theorem provers to determine whether a candidate is valid, i.e., whether it meets the specification. However, while MTL supports list and other data structures, the theory of lists in theorem provers is incomplete and thus insufficient to verify all possible summaries the synthesizer might generate. As a result, the synthesizers struggle to solve the synthesis problem described in Equation (3). To make synthesis tractable, we simplify the problem by first performing bounded synthesis, and subsequently sending candidate summaries that pass bounded synthesis to a general theorem prover to validate. In Equation (3), rather than searching for program summaries that satisfy the VCs for all program states (σ), we search only for a finite set of program states. For example, we limit all list data structures to lengths of up to size 2, and all integers to bit-widths of up to 7 bits. These parameters can be changed by the developers using METALIFT’s API. Note that we bound the program states only during the synthesis phase, and during verification we check if the synthesis phase output is true for all program states.

The summaries returned by the synthesizer using the target language and search space defined in Figure 11 and Figure 13, respectively, are shown in Figure 15. The output variable *count* is synthesized as a series of map and reduce operations. In the mapper phase, λ_m maps each word w in the input list to the key-value pair $(w, 1)$. In the reducer phase, *reduceByKey* groups all the unique keys in the map output, reduces each group using the add operator and returns a map containing the key-value pairs $(w, frequency)$.

After bounded synthesizer generates the summaries, they are parsed and represented using the constructs in MTL. Note that the generated summaries are not in the concrete syntax of Spark yet but it is expressed in the operators defined by the developers. Developers can easily convert these summaries using simple syntax-driven rules which we describe in Section 3.9. In the next section, we provide details about the verification of the synthesized summaries and invariants, as well as proving that the inferred invariant is indeed sound.

3.8 Verification

Given candidate summaries that are generated by the bounded synthesizer, METALIFT next automatically performs the full verification of the synthesized summaries and invariants. As previously stated, the synthesized summaries and invariants generated during the synthesis phase satisfy the verification conditions only for a finite set of program inputs. For instance, in our running example, the synthesis phase returns a solution that is only verified for lists with lengths up to 2 and, integers with bit-width up to 7. The goal of the verifier is to prove that the program summaries are valid for all the program states (all integers and lists sizes). METALIFT uses *satisfiability modulo theories* (SMT) solvers [10, 42, 7] to solve this verification problem. Formally, the theorem prover checks for the satisfiability of the following problem

$$\forall \sigma . \neg VC(S, ps, inv_1, inv_2, \dots, inv_n, \sigma) \quad (4)$$

```

def ps(counts, words):
    counts = reduceByKey(map(words,  $\lambda_m$ ),  $\lambda_r$ )
    return counts

def inv(j, count, words):
    return And((j  $\geq$  0), (j  $\leq$  length(words)),
              (count = reduceByKey(map(take(words, j)  $\lambda_m$ ),  $\lambda_r$ )))

def  $\lambda_m$ (v):
    return append(empty, Tuple(v,1))

def  $\lambda_r$ (v1, v2):
    return (v1 + v2)

```

■ **Figure 15** Program summaries and invariants generated by the synthesizer for the source code in Figure 2a.

where the definitions inferred during the synthesis phase are substituted for the placeholders for *ps* and *inv*'s in the VCs. Negating the assertion and checking if there exists some program state that satisfies the assertions is standard practice in program verification: if the theorem prover discovers a program state that satisfies the negated assertion then the generated program summaries or the invariants are incorrect. However, if there exists no such state then the inferred summaries and the invariants prove the validity of the verification conditions and thus the semantic equivalence of the summaries to the source code.

As mentioned in Section 3.3, MTL supports various data structures. METALIFT models list and tuples in MTL using the SMT solver's built-in functionality of algebraic data types. Algebraic data type definition requires the user to declare the data type and associate a sort (type) with the declaration. Following that, users declare the accessors and constructors for data retrieval and creation of new data structures, respectively. Associative maps in MTL are modeled using the the SMT solvers built-in theory of arrays. METALIFT generates the verification problem automatically by translating MTL to SMT-Lib format [9]. As the SMT-Lib format does not support higher order functions, we inline them while converting.

Figure 16 shows the simplified version of the verification condition generated during the analysis phase for the source code in Figure 2a. We now show that the invariant described in Figure 15 is necessary and sufficient to prove the verification conditions.

Initial Condition. The initial condition asserts that the loop invariant holds immediately before the loop. Before the loop executes, $j = 0$ and *counts* is an empty map. The invariant expresses *counts* as a series of map and reduce operation applied to the first j elements of the input words list. Since $j = 0$, the map reduce operation will be applied to an empty list, returning an empty map according to the definitions in Figure 11. Hence, the invariant holds in the initial state.

Preservation. The loop preservation VC asserts that if the loop invariant hold at any arbitrary iteration, j , of the loop then it should also hold in the next iteration, $j + 1$, of the loop. The notation $counts[w \mapsto e]$ denotes the assignments statement, i.e., the key w in *counts* gets assigned the value e . This is proved by induction. We first prove that the invariant holds at the initial condition, we assume that the invariant holds at iteration j , i.e., the map reduce operation has already computed the frequency of first j words in the list. We need to show that the invariant holds after one more execution of the loop. This is

true since in the $j + 1^{\text{th}}$ iteration of the loop, the map reduce operation would compute the frequency of the first $j + 1$ words from the input list, thereby incrementing the count of the $j + 1^{\text{th}}$ word in the output map *counts* by 1.

Termination. Finally, the termination condition states that if the loop terminates, the invariant should imply the post condition. This is true because at the end of the loop $j = \text{size}(\text{words})$ and the map reduce operation would have computed the frequencies for all the words in the list which is the same expression as the program summary.

Initial Condition	$\text{Inv}(j = 0, \text{counts} = \{\}, \text{words})$
Preservation	$\text{Inv}(j, \text{counts}, \text{words}) \wedge (j < \text{size}(\text{words})) \rightarrow \text{Inv}(j + 1, \text{counts}[\text{words}[j] \mapsto \text{words}[j] + 1], \text{words})$
Termination	$\text{Inv}(j, \text{counts}, \text{words}) \wedge \neg (j < \text{size}(\text{words})) \rightarrow \text{PS}(\text{count}, \text{words})$

■ **Figure 16** Verification conditions for the source code in Figure 2a.

3.8.1 Leveraging Additional Axioms

Verifying loop invariants in general is undecidable, as MTL supports recursion and higher-order functions. To aid in verification, developer can provide additional *axioms* to the theorem prover to prove the validity of loop invariants and program summaries for VCs such as in Figure 16. Figure 11 shows how developers can use MTL to provide operator specific axioms to METALIFT. These axioms are translated from MTL to SMT-Lib format and included in the verification problem by METALIFT. These axioms from our experience are simple properties like associativity, commutativity, and distributivity of the operators in the target language. If developers only need to ensure bounded correctness for the translation, they don't need to define these axioms.

Figure 17 shows the SMT-Lib translation of the map operator axioms described in Figure 11, where $\text{map_}\lambda_m$ is the inlined definition of the map operator. The first axiom states that the map operators is distributive over two input lists. While the second one asserts the inductive property of the map operator. Other properties described in Figure 11 can be similarly translated to SMT-Lib format.

```

1 ;distributive map lemma
2 (assert (forall ((lst1 (List T)) (lst2 (List T)) )
3 (= (map_λm (concat lst1 lst2)) (concat (map_λm lst1) (map_λm lst2))))))
4
5 ;inductive map lemma
6 (assert (forall ( (lst (List T)) (index U) )
7 (=> (and (>= index 0) (< index (length lst)))
8 (= (map_λm (tail lst index)) (concat (λm (get lst index))
9 (map_λm (tail lst (+ index 1))))))))))

```

■ **Figure 17** Translation of the axioms defined using MTL to SMT-Lib format.

3.9 Code Generation

The program summaries verified by METALIFT are expressed using the high-level operators defined by the developers. The final step is to implement syntax-driven rules to translate these summaries to the concrete syntax of the target DSL. This translation is much easier than the general translation from source language to target DSL API because the developers

- 1) only need to write rules for the operator they defined in the target language and
- 2) do not need to worry about semantic equivalence to the source code as METALIFT automatically verifies the program summaries.

In METALIFT, we conduct synthesis in the high-level MTL rather than searching through the concrete syntax of the DSL. This approach is used to make the search process more tractable during synthesis. A DSL may include multiple variations of the same operator, with each having minor differences while they are functionally equivalent. For instance, the Spark DSL contains multiple variations of the map operator (*map*, *flatMap*, *mapToPair*). Rather than searching through these different concrete implementations, we conduct the search using a single implementation that captures the different operators' high-level semantics. Once a solution is synthesized in the MTL, converting it to concrete syntax becomes straightforward. The developer can decide which variation to use on basis of the return value of the λ_m function. For example, if the λ_m function returns a key-value pair or list with single key-value pair, the developers can use *mapToPair* from the DSL. If the λ_m function returns a list containing multiple key-value pairs, the developers can then use *flatMapToPair*. Table 2 shows an excerpt of the translation rules for the Spark DSL. Applying the translation rules to the summary generated in Figure 15 would result in `words.mapToPair(v -> (v, 1)).reduceByKey((v1, v2) -> v1 + v2)`. Another advantage is that if the same operator can exist in multiple DSL (e.g. convolution in tensor processing libraries), we only need to perform the search once and then code generation rules can translate the summary into concrete syntax of any library. In Figure 18, we show the implementation for some of these rules for the Spark DSL.

■ **Table 2** Translation rules for converting program summaries in MTL to Spark DSL.

Pattern	Translation Rule
$\llbracket \text{map}(l, \lambda_m \rightarrow \text{list}(\text{Pairs})) \rrbracket$	<code>l.flatMapToPair($\llbracket \lambda_m \rrbracket$)</code>
$\llbracket \text{map}(l, \lambda_m \rightarrow (\text{Pair or list}(\text{Pair}))) \rrbracket$	<code>l.mapToPair($\llbracket \lambda_m \rrbracket$)</code>
$\llbracket \text{map}(l, \lambda_m \rightarrow T) \rrbracket$	<code>l.map($\llbracket \lambda_m \rrbracket$)</code>
$\llbracket \text{reduce}(l, \lambda_r) \rrbracket$	<code>l.reduce($\llbracket \lambda_r \rrbracket$)</code>
$\llbracket \text{reduceByKey}(l, \lambda_r) \rrbracket$	<code>l.reduceByKey($\llbracket \lambda_r \rrbracket$)</code>
$\llbracket \lambda_m(v) \rightarrow e \rrbracket$	<code>(v -> $\llbracket e \rrbracket$)</code>
$\llbracket \lambda_r(v1, v2) \rightarrow e \rrbracket$	<code>((v1, v2) -> $\llbracket e \rrbracket$)</code>
$\llbracket e_1 \text{ aop } e_2 \rrbracket$	<code>$\llbracket e_1 \rrbracket$ aop $\llbracket e_2 \rrbracket$</code>

4 Evaluation

We now describe our experience in building synthesis-driven transpilers using METALIFT. We have implemented the core of METALIFT in Python. We provide developers with a Python API for using the constructs in our MTL for defining the semantics of their DSL operators and search space description. METALIFT uses Rosette [41] as its synthesis back-end engine, and supports Z3 [42] and CVC5 [7] for verification.

In this section, we show that our MTL is general enough to be used to create compilers for different DSLs. We demonstrate this by creating compilers using our framework for two DSLs that target very different applications:

```

1 def codeGen(verifiedSummaries):
2   ps = verifiedSummaries['ps']
3   def eval(expr):
4     if expr[0] == "reducebykey":
5       return "%s.reducebykey(%s)"%(eval(expr[1]), eval(expr[2]))
6     elif expr[0] == "map":
7       if len(lm) == 1: map_func = "mapToPair"
8       else: map_func = "flatMaptoPair"
9       return "%s.%s(%s)"%(expr[1],map_func,eval(expr[2]))
10    ...
11   return eval(ps)

```

■ **Figure 18** Implementation of syntax-driven rules for translating summaries to executable code in target DSL.

1. **Distributed Computing DSL.** In this case study, we build a compiler that translates sequential Java code to Spark DSL [43]. Spark provides users with APIs to perform large-scale data processing efficiently by distributing the computations across multiple clusters.
2. **Hardware DSL.** In this case study, we build a compiler that translates Domino [35], which allows users to implement data-plane algorithms (such as congestion control and load balancing) for network switches to Banzai atoms, which represent atomic operations typically available in network switches. By combining these atoms, users can implement various programmable network switches with Banzai.
3. **Vector Operation DSL.** In this case study, we build a compiler that translates sequential C++ code to vector operations. For loops are generally slower compared to their vectorized operations on large datasets and libraries such as Scipy, Pytorch and Tensorflow provide efficient implementations of these vectorized operators.

These case studies differ in the program structure of the source code, in addition to targeting different domains. The Spark and vector case study contains programs with loops, whereas Domino contains only straight line programs with no looping constructs. Prior work [3, 35] implemented these case studies as two separate specialized compilers. We demonstrate that METALIFT can be used to create all these compilers, and that METALIFT simplifies the process of building DSL compilers that leverage program synthesis.

4.1 Case Study: Spark

MapReduce [15] is a popular programming paradigm which enables users to write compute intensive-applications capable of processing massive datasets by distributing the computations across multiple clusters. Mapreduce programming model decomposes the processing into two primitives *map* and *reduce*. MapReduce first breaks down the dataset into multiple independent chunks and then uses the following three stages to process the data:

1. **map phase:** each node in the cluster receives a small chunk of the data. Each node then locally processes the data by applying the mapper function and produces a set of key-value pairs.
2. **shuffle phase:** all the key-value pairs from different mapper functions are then sorted, grouped together by key and then redistributed to different nodes such that each node receives values belonging to the same key.
3. **reduce phase:** each node then aggregates the values using the reducer function.

Spark [43] is an open-source analytics framework. Spark provides users with highly-efficient implementations of map and reduce operations via its APIs in high-level languages such as Python and Java.

To leverage the optimizations provided by the MapReduce paradigm, Casper [3] and Mold [31] are two compilers that automatically translate legacy code written in Java to sequence of map and reduce operations. Casper used program synthesis to perform the transformations, whereas Mold used a syntax-driven rule based approach. We demonstrate that using METALIFT developers can build the core capabilities of the Casper toolchain by easily defining the semantics of the map and reduce using our MTL.

MetaLift’s implementation of Casper. We manually rewrote the Casper benchmarks from Java to C++, as METALIFT’s frontend currently do not support Java. We then define the semantics of the map and reduce operations using our MTL, as illustrated in Figure 11. We also provide additional axioms to assist the verifier in proving validity of the synthesized program summaries and loop invariants (Lines 23-39 in Figure 11). As shown in Figure 13, using MTL we provide the description of the search space for the program summaries, invariant, mapper and reducer functions. Casper implemented incremental grammar search to make the search tractable, and a cost-based evaluation model to select from different semantically equivalent program summaries. In our METALIFT implementation, we use incremental search and in Figure 14 we describe how developers can easily modify the driver code to implement such search strategies.

Results. We evaluate our implementation on the benchmarks [1] that were successfully translated by Casper. Our implementation translates **44** benchmarks of the **49** benchmarks that Casper translated. Synthesizer times out on the five benchmarks which our implementation failed to translate. We believe the reason for these failures is that METALIFT uses LLVM to generate verification conditions, whereas Casper implemented a specialized verification condition generator for Java source code directly, resulting in considerably more optimized VCs that synthesizers can easily solve. We use a timeout (synthesis + verification) of 60 minutes for all the benchmarks. The average running time for our implementation of the Casper benchmarks was \approx 3mins. We observed that METALIFT-generated code had the same structure to the ones which Casper synthesized (i.e., same number of map reduce stages and same implementation of the mapper and reducer functions), so we expect the output code to have similar performance as that generated by Casper. In contrast to Casper, which required **25578** lines of code, our implementation requires less than **1000** lines.

4.2 Case Study: Domino

Domino [35] is a domain-specific language for data-plane algorithms that run on programmable line-rate switches. The Domino DSL provides a C-like interface to define a “packet transaction” on a stream of network packets. While the syntax of Domino is a subset of C, there are a number of restrictions on memory allocation, loops, and control flow structure to prevent writing Domino programs which cannot be executed at “line-rate” (the speed with which packets arrive on a programmable switch). A particularly relevant restriction is that Domino is loopless, obviating the need to synthesize loop invariants as required in Spark. Figure 21 contains an example of a packet transaction written in Domino which implements a simple Rate-Control Protocol (RCP) [40] by accumulating the sum of packet round-trip-times (RTTs) for which the RTT is under the maximum allowable value.

```

1 StateResult atom_template(int state_1, int state_2, int pkt_1, int pkt_2) {
2   if (rel_op(Opt(state_1), Mux3(pkt_1, pkt_2, C()))) {
3     state_1 = Opt(state_1) + Mux3(pkt_1, pkt_2, C());
4   }
5   StateResult ret = new StateResult();
6   ret.result_state_1 = state_1;
7   ret.result_state_2 = state_2;
8   return ret;
9 }

```

■ **Figure 19** An example of a Banzai atom used in the Domino compiler [36].

Domino compiles to Banzai, a machine model for programmable line-rate switches. The Banzai target encodes hardware constraints fundamental to these switches, the most important of which is *atomicity*, thereby guaranteeing that packet operations occur transactionally. Banzai provides an abstraction over programmable switch architectures with the notion of an *atom*, a stateful processing unit that contains atomic operations which can be used to implement data-plane algorithms. An example of a Banzai atom is given in Figure 19.

Compilation to Banzai. The Domino compiler [35] has a three-stage pipeline to compile a feasible Domino program to Banzai. In the first stage, the compiler preprocesses the code by

- 1) recursively transforming branches into conditional assignments
- 2) rewriting operations on state variables to occur on temporary packet fields instead
- 3) converting to static single-assignment (SSA) form and
- 4) flattening to a three-address code representation.

In the second stage, the Domino compiler decomposes the input code into a sequence of “codelets”, a smaller block containing three-address code. To do so, the compiler executes dependency analysis on the input code to form a dependency graph. The compiler splits the input code into strongly-connected component blocks (“codelets”), and forms a meta-directed acyclic graph of these codelets, thereby capturing block-scale dependencies. The compiler then schedules the codelets in topological order to ensure that all dependencies are satisfied.

Finally, the Domino compiler performs code generation by

- 1) distributing work throughout codelets to ensure that no block takes too long (the “pipeline width” constraint) and
- 2) using the SKETCH [37] program synthesizer to map each codelet to one of a set of Banzai atoms which are feasible in hardware.

MetaLift implementation of Domino. We demonstrate how METALIFT can be used to significantly simplify the synthesis of Domino benchmarks to Banzai atoms. First, we encode the Banzai target language and grammar as a set of stateless operations, specified in Figure 20. By ensuring our target language and grammar contains only stateless operations, we guarantee atomicity. We then decompose the Domino benchmarks into codelets – which we represent as C++ functions – as in the second stage of Domino compiler. However, we make two key assumptions:

- 1) we assume that array reads and writes happen in between codelets to ensure our target language and grammar does not need stateful operations

```

1 def targetLang():
2   ## Variable and Constant Getters
3   def const(): return Choose(*CONSTANTS) # CONSTANTS is a list of allowed constants
4   def var(): return Choose(*VARS) # VARS contains the input to the codelet
5   def var_or_const(): return Choose(var(), const())
6   def opt(n): return Choose(n, 0)
7
8   ## Banzai Atoms
9   def arith_op(a, b): return Choose(a + b, a - b, a * b)
10  def rel_op(a, b): return Choose(a == b, a != b, a <= b, a < b, a > b, a >= b)
11  def raw(): return opt(var()) + var_or_const()
12  def rw(): return var_or_const()
13  def mul_acc(): return opt(var()) * var_or_const() + var() + var()
14
15  def pred_raw():
16    if rel_op(opt(var()), var_or_const()): return opt(var()) + var_or_const()
17    else: return var()
18
19  def if_else_raw():
20    if rel_op(opt(var()), var_or_const()): return opt(var()) + var_or_const()
21    else: return opt(var()) + var_or_const()
22
23  def sub(): ...
24
25  def nested_ifs(): ...

```

■ **Figure 20** The semantics of the target Banzai atoms defined using MTL.

```

1 #define MAX_ALLOWABLE_RTT 30
2 struct Packet { int size_bytes; int rtt; };
3
4 /* State variables */
5 int input_traffic_Bytes = 0;
6 int sum_rtt_Tr = 0;
7 int num_pkts_with_rtt = 0;
8
9 /* Transaction code */
10 void func(struct Packet pkt) {
11   input_traffic_Bytes += pkt.size_bytes;
12   if (pkt.rtt < MAX_ALLOWABLE_RTT) {
13     sum_rtt_Tr += pkt.rtt;
14     num_pkts_with_rtt += 1;
15   }
16 }

```

```

1 AddStateRet3(
2   # _input_traffic_Bytes
3   Add(_input_traffic_Bytes,
4     size_bytes),
5   # _sum_rtt_Tr
6   Add(rtt, _sum_rtt_Tr)
7   if Not(Ge(rtt, 30))
8   else _sum_rtt_Tr,
9   # _num_pkts_with_rtt
10  Add(_num_pkts_with_rtt, 1)
11  if Not(Ge(rtt, 30))
12  else Add(0, _num_pkts_with_rtt
13 )

```

■ **Figure 21** End-to-end synthesis of Domino DSL to Banzai via MetaLift.

- 2) and we assume that each codelet has up to three outputs, all stored in temporary packet fields, and that each later codelet in a “pipeline” receives the set of all relevant outputs from prior codelets.

Finally, we compile the decomposed Domino benchmark to LLVM IR using Clang, and then programmatically synthesize each codelet to the target language and grammar.

We show an end-to-end example of synthesizing a Domino benchmark in Figure 21. The METALIFT synthesized summary contains three atoms, two of which are `pred_raw` (see Figure 20 for atom definitions) and one is a stateless arithmetic operation. `AddStateRet3` is not an atom, but rather the language primitive to output three return values (the new state and packet variables) due to how it is represented the C++ Domino benchmark.

■ **Table 3** Comparison of *maximum* atoms per stage used in the synthesized output of the Domino compiler and METALIFT. The maximum atoms per stage for each benchmark are taken from the Domino paper [35].

Benchmark name	MetaLift atoms/stage	Domino atoms/stage
Bloom filter	3	3
Heavy Hitters	3	9
Flowlets	2	2
RCP	2	3
Sampled NetFlow	2	2
HULL	3	1
Adaptive Virtual Queue	3	3
Queueing priority computation	2	2
DNS TTL change tracking	2	3
CONGA	2	2

Results. We successfully synthesized all ten Domino benchmarks with our solution described above, which took a total of **1052** un-minified lines of code. For domino, the evaluation metric stated in the prior work [35] was not performance but rather feasibility, i.e., if we could transpile a program to a banzai atom then these are feasible to run on a programmable switch device. The average compilation time for these 10 benchmarks was ≈ 6 secs. The Domino compiler, measuring all C++ source files and headers, totals **4036** lines of code, so our solution leveraging METALIFT requires **74%** fewer lines of code. Table 3 shows that the synthesized output maximum atom count per stage from METALIFT is under the maximum number of atoms per stage that the Domino compiler used for all but one of the ten benchmarks. Differences can be partially attributed to different benchmark decompositions (and therefore, different numbers of stages): for example, in the “HULL” benchmark, the Domino compiler output had 7 stages with maximum 1 atom/stage while METALIFT had 5 stages with maximum 3 atoms/stage. The other primary source of variance is that the Domino compiler optimized for surface area and speed at the hardware level, while METALIFT only optimized for the number of atoms. As a result, certain benchmarks like “Heavy Hitters” were synthesized in far fewer atoms by METALIFT, but those atoms were slower at a hardware level. Nonetheless, the success of this case study further demonstrates the capability of METALIFT to decrease the complexity of verified lifting solutions while simultaneously raising the level of abstraction.

4.3 Case Study: Vector Operations

In machine learning workflows, one crucial step is to pre-process datasets, but writing processing pipelines using loops can be computationally expensive due to the large size of the datasets. To improve efficiency, libraries like Pytorch, Tensorflow, and Scipy offer highly optimized vector or matrix operations for performing these operations faster. To take advantage of the optimizations, we build a transpiler with METALIFT to translate loopy array processing programs to vectorized operations. For example, consider the program in Figure 23 that computes the sum of consecutive elements in an array. This sequential program can be implemented using a convolution operation with a kernel of $[1, 1]$ and a stride of 1. For this transpiler, we encode the semantics of the operators such as 1D convolution, element-wise vector (matrix) multiplication and dot-product using MTL. In Figure 23, we show the

definition of the 1D convolution operation in our MTL. Note that if we were to search for an equivalent convolution program using the concrete syntax of tensor libraries we will have to search and verify for each individual library. Instead, by lifting the semantics to our MTL, we need to perform this search only once and then using simple syntax-driven rules we can translate this to any of the tensor libraries such as Tensorflow or Pytorch.

Results. We evaluate our implementation on **5** array processing benchmarks which can be represented using the vector operations described above. These benchmarks are a combination of the stencil kernels introduced in prior work [20] and C++ kernels scraped from the web. We use the same 60 minutes timeout as the previous case studies. Since all these are loopy programs, METALIFT synthesizes any additional invariants also required to prove the functional equivalence. We can translate all the benchmarks with an average running time of ≈ 2 mins. Once we have the summaries synthesized in our MTL, we write code generation rules to translate them to PyTorch, Tensorflow and Scipy. Our implementation requires less than **500** lines of code.

```

1 vector<int> program(vector<int> data){
2     vector<int> result;
3     for (int i = 0; i < data.size() - 1; i++)
4         result.push_back(data[i] + data[i + 1]);
5     return result;
6 }
```

■ **Figure 22** Sequential C++ array processing program.

```

1 def conv(data, kernel, stride):
2     if length(1st) == 0 then empty
3     return prepend(dot_product(data, kernel), conv(tail(data, stride), kernel, stride))
```

■ **Figure 23** Semantics of the convolution operator using MTL.

5 Related Work

Program Synthesis-Based Compilers. Many tools have been developed previously which leverage program synthesis to translate code written in general-purpose programming languages to DSLs while preserving semantics. Examples of such compilers include STNG [20], which converts Fortran stencil computations to the Halide [32] DSL; QBS [14], which translates sequential Java database processing queries to SQL; Casper [3], which converts sequential Java code to map-reduce operations; and Domino [35], a compiler that translates network packet processing algorithms for programmable switches. All of these compilers are fully automated, but they are tailored to a specific DSL and cannot be reused to build a compiler for a new DSL. Implementing these tools required extensive knowledge of program synthesis and verification, making it difficult for developers to leverage the underlying approach of synthesis-based transformation for their own DSLs. METALIFT, on the other hand, uses MTL to provide a very high-level abstraction for developers to specify the semantics of their DSLs and automatically build compilers that can perform semantic preserving transformations. Other prior work [34] leverages program synthesis to perform superoptimization for the x86

instruction set and [23] employs a data-driven approach for verifying peephole optimization in LLVM. These tools perform transformations for low-level languages while METALIFT builds compilers for DSLs.

Syntax-Driven Transpilation. The traditional solution to the translation problem is to create a syntax-driven compiler. These rule-based systems rely on users to create rules that pattern match in order to perform the required translation. An example of one such compiler is [31], which translates sequential Java code to into MapReduce operations. As such, these rule-based systems tend to be difficult to design and are vulnerable to translation errors. Perhaps even more problematic is that such approach is brittle to changes in DSL semantics.

Neural Approaches for Transpiling. Recently, there has been lot of interest in using neural machine translation to perform source-to-source translations. A number of approaches, including supervised [25, 13] and unsupervised [33] learning based techniques have been proposed for translating between general-purpose programming languages. These models do not require any input from developers, but require massive amounts of data for training. For instance, [33] required over 100 million functions from C++, Python and Java to train their model. METALIFT targets DSLs for which such huge amounts of data may not be available always. Furthermore, the transformations performed by these neural models are not verified, which may provide an opportunity to introduce bugs in the code.

6 Conclusion and Future Work

In this paper we described METALIFT, a unified platform for building DSL compilers. METALIFT allows developers to build compilers for their own DSLs by utilizing the synthesis-based program transformation approach that has been the underlying technique for many previous DSL compilers [2, 20, 35]. METALIFT achieves this with its design of a specification language called MTL. Using MTL developers can express the semantics of their target DSL and search space description to guide the synthesis engine. This programmatic approach to describing DSLs allows METALIFT to build compilers for various DSLs. We described our experience in building synthesis-driven transpilers by using METALIFT to build three transpilers targeting very different application domains. We demonstrate that METALIFT significantly reduces the effort required to create specialized implementations of these three compilers. With a unified framework, METALIFT opens up new research directions for automated transpilation, such as leveraging dynamic execution traces to automatically infer loop invariants, and improving synthesis with neuro-symbolic methods, which can learn effectively search strategies from similar programs, as well as oracle-guided synthesis methods [28, 29], which integrate verification with synthesis and extend the expressive power of synthesis queries beyond SMT. METALIFT is modular to easily add any of these optimizations. As a framework, METALIFT has the potential of dramatically lowering the barrier for both research into synthesis driven transpilation and adoption of new specialized high performance hardware and DSLs.

References

- 1 Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. <https://github.com/uwplse/Casper/tree/master/bin/benchmarks>, Accessed: 2022-03-14.
- 2 Maaz Bin Safeer Ahmad and Alvin Cheung. Leveraging parallel data processing frameworks with verified lifting. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, pages 67–83, 2016.

- 3 Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1205–1220. ACM, 2018.
- 4 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.
- 5 Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michal Moskal, and Nikhil Swamy. Calibrating research in program synthesis using 72,000 hours of programmer time, 2013.
- 6 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimantas Avizienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference, DAC '12*, pages 1216–1225, New York, NY, USA, 2012. ACM. doi:10.1145/2228360.2228584.
- 7 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.
- 8 Mike Barnett and Rustan Leino. Weakest-precondition of unstructured programs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87. ACM Press, September 2005. URL: <https://www.microsoft.com/en-us/research/publication/weakest-precondition-of-unstructured-programs/>.
- 9 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.
- 10 Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 33, pages 1267–1329. IOS Press, second edition, 2021.
- 11 Rastislav Bodík and Barbara Jobstmann. Algorithmic program synthesis: introduction. *STTT*, 15(5-6):397–411, 2013.
- 12 Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 363–375, 2010.
- 13 Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18*, pages 2552–2562, Red Hook, NY, USA, 2018. Curran Associates Inc.
- 14 Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 3–14, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462180.
- 15 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. doi:10.1145/1327452.1327492.
- 16 Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, July 2008. doi:10.1109/MM.2008.57.
- 17 Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 13–24, 2010.

- 18 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 343–361, Cham, 2015. Springer International Publishing.
- 19 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.
- 20 Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726, New York, NY, USA, 2016. ACM. doi:10.1145/2908080.2908117.
- 21 David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszal, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators, 2018.
- 22 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: <http://dl.acm.org/citation.cfm?id=977395.977673>.
- 23 David Menendez and Santosh Nagarakatte. Alive-Infer: Data-driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 49–63, New York, NY, USA, 2017. ACM. doi:10.1145/3062341.3062372.
- 24 Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 225–242, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359641.
- 25 Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Lexical statistical machine translation for language migration. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 651–654, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491411.2494584.
- 26 Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: <https://openreview.net/forum?id=rJ0JwFcex>.
- 27 Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM. doi:10.1145/2594291.2594339.
- 28 Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeuffer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. UCLID5: multi-modal formal modeling, verification, and synthesis. In *34th International Conference on Computer Aided Verification (CAV)*, volume 13371 of *Lecture Notes in Computer Science*, pages 538–551. Springer, 2022.
- 29 Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. Satisfiability and synthesis modulo oracles. In *Proceedings of the 23rd International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2022.
- 30 Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *OOPSLA 2015 Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, October 2015. URL: <https://www.microsoft.com/en-us/research/publication/flashmeta-framework-inductive-program-synthesis/>.

- 31 Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, pages 909–927, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660228.
- 32 Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 519–530, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462176.
- 33 Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: <https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html>.
- 34 Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA – March 16 – 20, 2013*, pages 305–316, 2013.
- 35 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 15–28, 2016.
- 36 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Domino examples, October 2018. URL: https://github.com/packet-transactions/domino-examples/blob/master/banzai_atoms/pred_raw.sk.
- 37 Sketch. <https://people.csail.mit.edu/asolar/>, 2016. Accessed: 2016-05-01.
- 38 Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008. AAI3353225.
- 39 Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014. doi:10.1145/2584665.
- 40 C.-H. Tai, J. Zhu, and N. Dukkipati. Making large scale deployment of rcp practical for real networks. In *IEEE INFOCOM 2008 – The 27th Conference on Computer Communications*, pages 2180–2188, 2008. doi:10.1109/INFOCOM.2008.285.
- 41 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013*, pages 135–152, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509586.
- 42 The z3 theorem prover. <https://github.com/Z3Prover/z3>, 2017.
- 43 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12*, 2012.

Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

Sarah Harris ✉

University of Kent, Canterbury, UK

Simon Cooksey ✉ 🏠 

University of Kent, Canterbury, UK

Michael Vollmer ✉ 🏠 

University of Kent, Canterbury, UK

Mark Batty ✉ 🏠

University of Kent, Canterbury, UK

Abstract

Memory safety issues are a serious concern in systems programming. Rust is a systems language that provides memory safety through a combination of a static checks embodied in the type system and ad hoc dynamic checks inserted where this analysis becomes impractical. The Morello prototype architecture from ARM uses capabilities, fat pointers augmented with object bounds information, to catch failures of memory safety. This paper presents a compiler from Rust to the Morello architecture, together with a comparison of the performance of Rust's runtime safety checks and the hardware-supported checks of Morello. The cost of Morello's always-on memory safety guarantees is 39% in our 19 benchmark suites from the Rust crates repository (comprising 870 total benchmarks). For this cost, Morello's capabilities ensure that even unsafe Rust code benefits from memory safety guarantees.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software safety; Software and its engineering → Object oriented languages

Keywords and phrases Compilers, Rust, Memory Safety, CHERI

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.39

Category Experience Paper

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.25>

Software (Source Code): <https://github.com/kent-weak-memory/rust>

archived at `swh:1:dir:966327cc0ecb3fb4d2196b6f0912d775392fafa5`

Funding This work has been supported by the EPSRC under the DSbD Software Ecosystem grant programme EP/X021173/1.

Acknowledgements This paper was greatly improved thanks to the responses of anonymous reviewers. We extend our thanks to Jessica Clarke for her invaluable help with CHERI LLVM.

1 Introduction

Low-level programming entails delicate use of memory in a setting where common mistakes can lead to serious bugs and security vulnerabilities in critical code. *Memory safety* is the absence of these errors – where only correctly allocated regions of memory are accessed and freed. Unsafe uses of memory are the most critical software flaws today, they create security vulnerabilities, and they are widespread: out-of-bounds writes are the most dangerous security flaw in the Mitacs Common Weakness Enumeration [8]; Microsoft found that 70%



© Sarah Harris, Simon Cooksey, Michael Vollmer, and Mark Batty;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 39; pp. 39:1–39:27



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



of the security bugs in Windows were as a result of unsafe use of memory [23]; and in the Chromium browser 36% of bugs are caused by use-after-free errors [33], with a further 33% stemming from other unsafe uses of memory [33].

It is possible to automatically enforce memory safety, avoiding all the attendant bugs and security flaws. This enforcement comes at a cost, however, and how exactly the cost is levied is a design choice. In this paper we compare the runtime performance of the mechanisms that enforce memory safety in *Rust* and in the *Morello* architecture, and ultimately combine them, improving the coverage of Rust's memory safety guarantee.

Rust provides a guarantee of memory safety to all well-typed code. Much of the cost of this guarantee is handled by a static analysis in the type system, but it also uses runtime checks when proving safety statically would be costly or impossible. One can forgo the safety guarantee and its checks by designating a block of code as **unsafe**: memory accesses within this block are not required to pass the full rigour of the type system. Unsafe code is used sparingly for interoperability with non-Rust components and for performance. **unsafe** annotations highlight code where memory errors can survive.

Rust alleviates memory errors in common code while remaining flexible enough to support systems programming through the provision of **unsafe** blocks. Rust imposes two costs: programmers must adhere to a more restrictive type system, and there are runtime costs to support the safety guarantee. The combination of safety, pragmatism, and performance is why Rust is now the official second language of the Linux kernel alongside C [24].

Morello is a prototype ARM processor that provides *capabilities*: fat pointers, augmented with permissions and bounds information. Morello processors use this metadata to enforce memory safety at run time, halting programs when safety is violated – for example if a program makes an out-of-bounds memory access. Programs that protect every memory access are described as *Purecap*, but one can forgo the safety checks by accessing memory in *Hybrid* mode. In contrast with Rust, full-scale systems programming is possible in Purecap mode: there are Purecap Morello ports of BSD, Linux and Android [34, 4, 3]. Programming in Purecap mode ensures memory safety from the first, albeit with the possibility of runtime errors where safety would otherwise be violated. Further development effort improves the stability of the system.

The safety guarantees provided by Rust and Morello are subtly different. Morello's capabilities track only approximate bounds information (see §2.3), recording the size of objects as a floating-point number; Rust applies compiler optimizations to the runtime checks that it emits, removing unnecessary checks and improving performance. Even so, Rust and Morello provide similar guarantees that may be used in a complementary way. Purecap-Morello will check pointers are used safely even in **unsafe** blocks, for example. This is where Rust and capability hardware mesh neatly, where Rust cannot validate the safety of memory use the underlying Morello hardware can.

In this paper we will explore the interplay between the Morello prototype hardware and the Rust programming language with a focus on runtime performance. We find the performance cost of always-on hardware memory safety checks to be quite high but not prohibitively so, and the design philosophies of Rust's static memory safety guarantees and Morello's dynamic memory safety to be well-matched. This comparison will also serve as a way to benchmark the real-world performance of improvements to the Morello architecture in the context of cutting edge memory-safe programming practices. We present the following contributions:

1. A Rust compiler that targets the Purecap- and Hybrid-mode Morello prototype capability hardware (§3).
 - This includes a reasoned choice for the semantics of `usize` on a machine where pointers are not just integers (§3.1).
2. A ported Rust standard library with fixes to incorporate capabilities (§3.6).
3. A performance analysis of Rust on Morello (§4, §5):
 - Our benchmark suite of 19 crates from the Rust crates repository (§4.6, Appendix A).
 - Analysis of the benefits of bounds checking elision on capability hardware (§5.1).
 - Comparison of Rust on Hybrid-Morello to Rust on Purecap-Morello (§5.2).
4. Our artefacts which will be made public upon publication.

2 Background

For a language to provide practical memory safety, it must present a usable interface to the programmer and have acceptable performance. *Memory safety* comes with a number of requirements:

- ① values must be initialized before reading, especially pointers,
- ② values must not be accessed after deallocation,
- ③ reads and writes must be within the bounds of an object’s memory allocation,
- ④ values must be deallocated exactly once.

There are a number of approaches to enforcing these requirements: from garbage collection as used by Java, Go, OCaml and many others; assorted static and dynamic validators – Rust is one such system, using linear types, lifetimes and dynamic bounds checking; and now hardware schemes like Morello.

2.1 Rust

Rust [21] is a relatively young programming language, version 0.1 was released in 2012 [27], and is notable for incorporating a number of features geared to providing memory safety, covering the desiderata above. Minimising runtime cost while providing powerful features to programmers is a central design aim for current Rust [35], so the majority of these features are applied statically during compilation. There are however still some cases where it is regarded as impractical to infer bounds statically.

The most relevant features to Rust’s memory safety are [18, 28]:

- uninitialised values are not normally¹ allowed ①;
- move semantics, the `Drop` trait, and references prevent access to deallocated values ②;
- array and slice indices are the only pointer arithmetic normally¹ available, and these are bounds checked at runtime ③;
- move semantics, the `Drop` trait, and careful API design protects against double free ④;
- move semantics and the `Drop` trait provide some protection against memory leaks by making the default behaviour that objects are freed when their lifetime ends ④, but can be defeated by functions like `std::mem::forget()` and `Box::leak()` and by other issues²;

The most important of these safety features is the combination of move semantics, the `Drop` trait, references, and lifetimes, which together provide much of Rust’s memory safety guarantees.

¹ In `unsafe` it is possible to break these conditions, but this is not the default.

² Reference counted pointers can leak if used improperly, and some edge cases in exception handling can cause `Drop` not to execute.

2.1.1 Move and Drop

The simplest way to allocate memory in Rust is to use the stack. This works much like C, with a value being allocated memory on entry to a block, and deallocated on exit when the stack frame is popped:

```
struct Data { a: i32, b: i32 }
fn automatic_memory() {
    // data automatically allocated on the stack here:
    let data = Data{a: 1, b: 2};
    // ...
    // data falls off stack here
}
```

Large values and data with a lifetime that doesn't match that of a scope require dynamic memory allocation, which in C would be provided via `malloc()` and `free()`. In Rust this is provided using a combination of move semantics and the `Drop` trait. `Drop` allows a type to provide a method that will be called automatically when an instance of it leaves scope, which means that memory allocations can be managed semi-automatically by so-called “smart pointers”. The simplest implementation of this in Rust is the standard library type `Box`, which allocates memory on the heap when instantiated and uses `Drop` to ensure that it will be deallocated when the `Box` leaves scope. The `Box` value itself acts as a handle, tracking the lifetime of the allocation and providing access to the allocated memory while remaining a technically separate value.

```
fn heap_memory() {
    // data allocated on the heap here:
    let data = Box::new(Data{a: 1, b: 2});
    // ...
    // Box falls out of scope here, heap allocation automatically freed
}
```

Move semantics expand the utility of this approach by allowing the `Box` to be moved to a different name or out of scope, while preventing it from being duplicated or deallocated, which might otherwise cause the allocation to be freed twice or left to leak [4](#).

```
fn inner_scope() -> Box<Data> {
    // data allocated on the heap here:
    let data = Box::new(Data{a: 1, b: 2});
    // ...
    data // data is moved out of the function here
}
fn moved_box() {
    // data moved to outer scope here:
    let data = inner_scope();
    // ...
    let new_name = data; // data moved to new_name here
    assert!(data.a == 1); // compiler error: use of moved value: `data`
    assert!(new_name.a == 1); // ok
    // data falls out of scope here, heap allocation automatically freed
}
```


`Box` covers a wide range of use cases, and the same general design can be used to build more complex, more powerful tools to cover more demanding problems. The standard library provides a number of options, including dynamically resizable arrays via `Vec`, and reference counted allocations via `Rc` using the same mechanism.

2.1.2 References and lifetimes

While move semantics and `Drop` cover many uses of dynamic memory allocation, values can only be in one place at a time, and the resulting passing around can quickly become inconvenient. The solution to this problem is Rust’s reference types. These behave similarly to ordinary pointers, but enforce extra rules that are checked by the compiler ¹:

- a reference must point to a value, i.e. there are no null references ①,
- values pointed to must be currently allocated and correctly aligned ①,
- a value can either be referenced once mutably, or multiple times immutably,
- values can only be mutated via a mutable reference ², and
- values cannot be moved or deallocated while referenced ②.

The compiler statically checks these rules using a system of inferred lifetimes, which allow references to be moved around and copied in non-trivial ways while maintaining safety. This concept has its roots in region-based memory management [14, 11], and it prevents access to values after deallocation while still providing power and flexibility ②.

In the example below, the two `&mut data` references are not permitted to have overlapping lifetimes. Rust infers that an object’s lifetime ends when the last reference to it goes out of scope – in this example, at the end of `referenced_box()`.

```
fn referenced_box() {
    // data allocated on the heap here:
    let mut data = Box::new(Data{a: 1, b: 2});
    // two immutable references exist during this call:
    use_data(&data, &data);
    // and two mutable references, which causes a compiler error:
    // cannot borrow `data` as mutable more than once at a time
    use_data(&mut data, &mut data);
    // reference only exists for duration of expression:
    *get_field(&mut data) = 3;
    // no references exist by here, so data freed without errors
}

fn get_field(data: &mut Data) -> &mut i32 {
    // compiler infers lifetime of return from argument
    &mut data.a
}

fn use_data(a: &Data, b: &Data) {
    assert!(a.a == b.a);
}
```

Dangling references, i.e. references to values which would be de-allocated at the end of a scope, are forbidden in Rust.

¹ Rust doesn’t currently have a formal specification, so the best sources for this information (besides the compiler source code) are the Rust Book [18] and the Rust Reference [28]

² Though this can be circumvented via a mechanism called “interior mutability”.

```
fn dangle() -> &u32 {
    let value = 0;
    &value
}
// this function's return type contains a borrowed value, but there is no
// value for it to be borrowed from
// help: consider using the `static` lifetime
```

Rust instead provides a mechanism for overriding the point at which a lifetime ends by explicit annotation:

```
fn dangle2<'a>() -> &'a u32 {
    let value = 0;
    &value
}
```

The *borrow checker* is the name of the machinery in Rust which statically detects invalid uses of references and keeps track of when objects' lifetimes end.

2.1.3 unsafe

Rust includes a mechanism that allows programmers to choose to break the rules described above. This is useful for performance-critical hand optimisation and working around the limitations of the compiler's static checking. Use of `unsafe` indicates that something odd is afoot, it should be used sparingly and serves as an explicit marker to signal to programmers and auditing tools alike that these pieces of code require additional scrutiny.

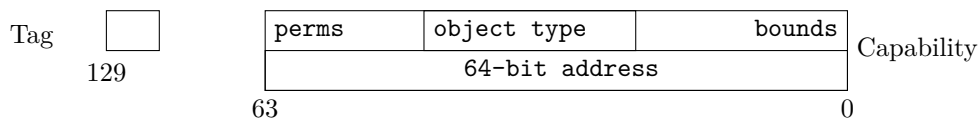
`unsafe` isn't a free pass to do anything – only a very specific set of extra privileges are available within these sections (see the Rust Book [18, §19.1]):

- ordinary C-like pointers called *raw pointers*, which don't have the limitations of references, can be dereferenced ~~1~~ ~~2~~ ~~3~~;
- functions and types marked `unsafe` can be used, allowing access to additional library APIs ~~1~~³ ~~2~~⁴ ~~3~~⁵ ~~4~~⁶;
- static variables can be accessed (which come with thread safety issues);
- traits marked `unsafe` can be implemented, automatically creating more `unsafe` code;
- unions can be accessed, which can be used to bypass type checking ~~1~~ ~~3~~;
- external C/C++ functions may be called, importing the memory safety concerns of those languages ~~1~~ ~~2~~ ~~3~~ ~~4~~.

In return, the programmer promises not to break any of the language's invariants.

The most important of these are the ability to use raw pointers and to call `unsafe` interfaces. Raw pointers are the most significant hole in the safety guarantees that Rust can provide, but *they are the primary point of compromise between full safety and a usable systems programming language*. Raw pointers are motivated by interoperability with non-Rust components, either through the operating system ABI or through linking C components into Rust programs, or Rust components into C programs. Raw pointers allow all the usual trickery, including creating them from integers, arbitrary arithmetic, null pointers, dangling

³ `std::mem::MaybeUninit`
⁴ `std::slice::from_raw_parts()`
⁵ `slice::get_unchecked()`
⁶ `Box::from_raw()`

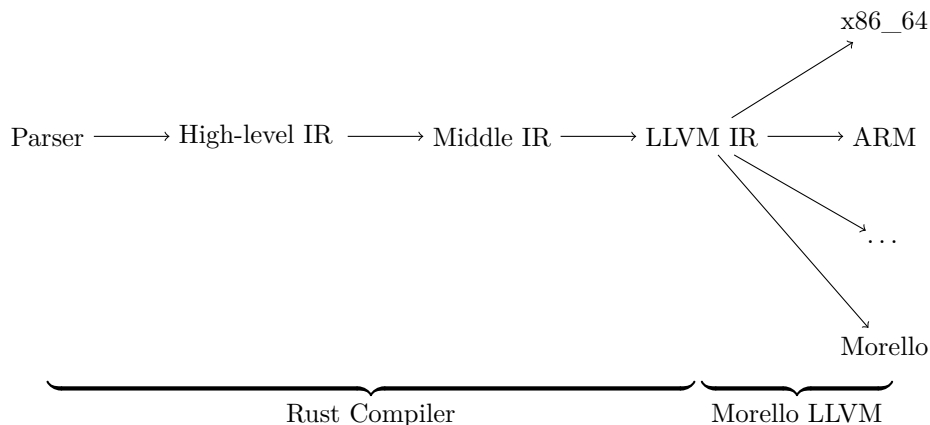


■ **Figure 1** The structure of a 128-bit capability.

pointers, freedom to cast between types, and so on. Access to `unsafe` interfaces enables use of a number of standard library features, notably `std::mem::transmute()`, which allows casting between any pair of types with the same size, and `std::mem::MaybeUninit`, which allows the creation of uninitialised values [\[1\]](#).

2.2 The Rust Compiler

The Rust compiler is mostly self-hosting, i.e. Rust is implemented in Rust. The front-end, type checking, middle intermediate representation (MIR), and an increasing number of optimisations are implemented in Rust. LLVM provides the backend and remaining optimisations, consuming LLVM IR and compiling to a range of targets including ARM and `x86_64`. This is all very convenient for porting Rust to Morello, as there is an existing LLVM implementation for Morello [20]. A sketch of the compiler’s structure is drawn below.



The majority of the compiler changes outlined in §3 are over the Middle IR portion of the compiler.

2.3 Capability hardware

CHERI is a generic instruction set extension to introduce *capabilities*, an extended pointer representation designed to add hardware security to memory accesses [36]. Capabilities add validity and permission information to pointers, expanding them to 128-bits on 64-bit platforms. Each capability is attached to a 1-bit *validity tag*, and this is required to be set to successfully perform memory accesses via the capability. Further, valid capabilities can only be constructed from other valid capabilities, and only in ways which don’t exceed the permissions of the parent capability. This property allows software to be separated into *compartments* which have limited, controlled access to one another.

Capabilities add four pieces of metadata to plain pointers: bounds, permissions, flags, and object type. An example of the structure of a 128-bit capability can be seen in Figure 1.

Bounds detail the range of memory which a capability is allowed to address; the hardware uses this information to perform automatic bounds checking. To be effective, the bounds of capabilities must be appropriately restricted to the object or buffer they point to. The

CHERI LLVM project extends the LLVM compiler infrastructure to do exactly this [20]. Most C and C++ programs, as well as a number of other languages with an LLVM backend, benefit because they automatically gain bounds checking with minimal to no modifications. It is worth noting, however, that CHERI's bounds checking has a limitation: to keep capability sizes reasonable, bounds information uses a floating point encoding. This means that bounds for larger regions become increasingly approximate, and marginally out of bounds accesses may succeed if regions are not padded to fill the extra space.

Permissions can be used to provide fine-grained read, write, execute, and capability manipulation protections for each capability.

Flags provide space for architecture-specific controls for each capability.

Object types are used as part of a mechanism to *seal* capabilities, such that they can only be dereferenced by specifically and deliberately chosen pieces of code. This opens up possibilities for compartmentalising programs and moving data and permissions through untrusted compartments without degrading security properties.

As a result of the requirement for capabilities to be derived from other valid capabilities, the provenance of each capability must be well-defined. This is a critical part of controlling the access rights of different parts of programs. For these limitations to hold, validity bits in uninitialised memory must be cleared before access is granted. This is expected to be provided by hardware or low level system software [37, §3.6.2]. This ensures that any uninitialised pointer will be invalid, and therefore impossible to dereference accidentally. The downside of the provenance property is that integer-to-pointer casts in existing code are likely to become invalid, though the changes needed to fix this are often minor. This is unsurprising given that integer-to-pointer casts are something of a headache for provenance analyses already [22].

Interpreted generously, bounds checking and provenance guarantees cover points ❶ (no use of uninitialised pointers) and ❸ (bounds checking) of §2. Research into ways that CHERI might be used to provide temporal safety guarantees is ongoing [39]. There is research investigating implementations of `free()` that can sweep the memory and invalidate any pointers into the free'd memory region [41] ❷. Even with temporal safety, memory leak bugs remain a problem to be solved by other tools ❹.

2.3.1 Morello prototype

Morello is a prototype platform for exploring capabilities, and is based on an application-class ARM SOC, the same sort that is found in modern smartphones and ARM-based personal computers. It is supported by a suite of open source software, including a C/C++ compiler based on LLVM [20], a FreeBSD port [34], and custom build automation tooling [9]. Software for Morello can be compiled in two different modes, both of which are supported by the CHERI BSD operating system we use [34].

Hybrid mode

In this mode of compilation, pointers are stored using a plain integer representation instead of capabilities, and normal ARM load and store operations are used to dereference them. Pointers are transparently restricted by the bounds of a single default capability provided by the operating system, with software running as if it were using normal ARM hardware. Rust code compiled with the non-CHERI compiler target triple `aarch64-unknown-freebsd` will run on CHERI BSD in this mode. While this mode is defined by the use of capability unaware load and store operations it is still possible to explicitly use capabilities, if the programmer desires and the facility is exposed by the language.

Purecap mode

In *pure capability* (Purecap) mode, all pointers are represented using capabilities, and capability instructions are used to access memory. Our modified compiler adds a new target triple to support this mode called `aarch64-unknown-freebsd-purecap`. This target configures the LLVM backend to emit capabilities instead of plain pointers, and the Rust compiler to use data layouts appropriate for capabilities. The details of the changes necessary to enable Rust for Purecap Morello are outlined in the next section.

Morello uses 128-bit capabilities, throws hardware exceptions if an invalid capability is dereferenced, and the maximum bounds size precisely representable is 4 KiB [1].

3 Adjustments to the Rust compiler and standard library

We describe the changes to accommodate capabilities in the Rust compiler. Recall that capabilities are 128-bit with one invisible tag bit which is maintained transparently by the hardware. To be able to use capabilities to represent pointers, a number of modifications to the compiler and standard library are necessary. Our port of Rust is based on release 1.56.0 (Edition 2021 Rust).

3.1 Rust semantics open question: `usize`

`usize` is an integer type, ambiguously defined by the documentation as the “[...] pointer-sized unsigned integer type.” [26] This is a straightforward definition on conventional architectures with integer pointer representations, but on Morello the meaning becomes unclear. There are two obvious interpretations for the semantics of `usize`:

- `usize` should be an integer and contain only an address, i.e. the lower word of a capability – a 64-bit number, or
- `usize` should behave like an integer and contain a whole capability, i.e. a Morello double-word sized 128-bit number.

We chose to explore the 64-bit approach for a number of reasons. First, the Rust community have sought to resolve this, and there are ongoing discussions that are leaning towards word-sized `usize` [31, 25]. Secondly, the previous work by Sim [32] explored 128-bit `usize` and was left with a handful of technical limitations which would be side-stepped by using machine-word-sized `usize`. Finally, there are several technical benefits to 64-bit `usize`, which we describe below.

Efficiency

`usize` is the only type of integer that can be used in array indexing, and is also used to represent lengths of arrays and sizes of types. These uses are very common, and only require that `usize` be able to represent the full range of addressable memory locations. In comparison, pointer-integer casts that would only function if `usize` stored a complete capability are rare. Making `usize` large enough to hold a capability would leave extra space that would be wasted in the vast majority of uses.

Robustness

While allowing `usize` to hold a valid capability would let simple pointer-integer-pointer round-trip casts work unmodified, it would also introduce inconsistent behaviour in many other cases. The tighter provenance model applied by CHERI invalidates capabilities derived only from integers, and also those produced by many arithmetic operations, including bitwise

39:10 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

logic. Given that bitwise logic is a common use case for integer-pointer casting, this would likely cause many unexpected capability faults. Ensuring instead that no `usize` can hold a valid capability guarantees that the compiler will always flag these cases, saving confusion and debugging time. This is traded against the cost of needing to make minor changes to simpler cases that might otherwise have worked unmodified, which we believe to be an acceptable sacrifice. Making this choice negates the major advantage of a 128-bit representation: being able to hold a valid capability.

Data compatibility

Under architectures currently supported by Rust, a cast from pointer to `usize` is expected to yield an integer containing the address being pointed to. Capabilities contain more information than this. Making `usize` a 64-bit integer containing only the address portion of the capability retains the expected behaviour.

3.2 Target specification

The Rust compiler has records of the size of various types for each platform it supports. This includes the size of pointers, which also decides the size of `usize`. To support Morello Purecap mode, the compiler needs more fine-grained information about the layout of pointers, and the size of `usize` needs to be decoupled from the in-memory size of pointers.

To do this, we have implemented *pointer width* and *pointer range*, where the compiler previously only had a single pointer size. Under mainstream architectures, these two values are all equal and redundant, but on Morello they are differentiated. *Pointer width* describes the in-memory size of pointers. Under Purecap, this will be the total size of a capability (128-bit), excluding the validity tag which is stored separately by the hardware. *Pointer range* describes the size of the address portion of pointers. Under Purecap, this will be the size of a plain pointer (64-bit), and the subset of a capability that contains the target address. Pointer range will also be the size of `usize`.

```
1 pub fn target() -> Target {
2     Target {
3         llvm_target: "aarch64-unknown-freebsd".to_string(),
4         pointer_range: 64,
5         pointer_width: 128,
6         data_layout: /* ... */,
7         arch: "aarch64".to_string(),
8         options: TargetOptions {
9             features: "+morello,+c64".to_string(),
10            llvm_abiname: "purecap".to_string(),
11            max_atomic_width: Some(128),
12            atomic_pointers_via_integers: false,
13            merge_functions: MergeFunctions::Disabled,
14            ..super::freebsd_base::opts()
15        },
16    }
17 }
compiler/rustc_target/src/spec/aarch64_unknown_freebsd_purecap.rs:3
```

■ **Figure 2** The Rust target options for the target triple `aarch64-unknown-freebsd-purecap`.

Information about a target is stored within the compiler using the structure shown in Figure 2. The target specification defines fundamental properties of the architecture and operating system. Morello inherits its base properties from the Aarch64 FreeBSD target, and then overrides some specifics. The new pointer entries can be seen on lines 4 and 5. Line 6 gives the standard data layout string describing the Morello architecture to LLVM, for brevity we do not expand on the details of this. For Purecap, this uses an extension specific to Morello LLVM [20] to specify that pointers be stored in address space 200, meaning that they should be represented using capabilities. Lines 9 and 10 specify the Purecap ABI, and are required to enable relevant features in Morello LLVM. Lines 12 and 13 disable some optimisations that are not yet compatible with Morello.

3.3 Constant evaluation

An unexpected source of problems for our changes to the compiler was Rust’s constant evaluation feature. Constant evaluation allows a subset of Rust expressions to be interpreted during compilation, as is demonstrated by the snippet below.

```

1  const MAGIC: u32 = long_multiply(3, 5)*7;
2  const fn long_multiply(a: u32, b: u32) -> u32 {
3      let mut a_shifted = a;
4      let mut b_shifted = b;
5      let mut result = 0;
6      while a_shifted != 0 {
7          if a_shifted & 1 == 1 {
8              result |= b_shifted;
9          }
10         a_shifted >>= 1;
11         b_shifted <<= 1;
12     }
13     result
14 }
```

This snippet contains only `consts`, and the compiler will fully evaluate the value of `MAGIC` at compile time. `long_multiply(3,5)` will be evaluated to 15, and then `MAGIC` will be evaluated to 105. The constant evaluator has an internal representation of memory so that it can run constant code even when it contains mutable values, as on lines 3, 4, and 5 of this example. Constant evaluation uses the same data layout as the rest of the compiler, and the subset of the language allowed includes support for pointers. Capabilities add extra non-address components to pointers, so constant evaluation must be modified to take these into account. While it might be possible to enforce the full set of capability rules during interpretation, we currently believe that Rust’s semantics already enforces them. For the time being we simply leave the unused space uninitialised, but the changes needed to the interpreter are still wide-reaching.

Values during interpretation can be represented either as large contiguous allocations, or single values represented directly⁷. Memory allocations are represented as arrays of data bytes, with auxiliary information about which bytes have been initialised. The compiler relies on type information to describe the structure of the data contained in the allocation.

⁷ Single values are handled separately as a performance optimisation.

39:12 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

Memory allocations themselves remain unchanged, the unused metadata bytes of capabilities are left uninitialised. References to subsets of memory allocations are passed around inside the compiler as the `AllocRange` type, so this must be extended to include width and range information. Without the extra information, the compiler does not know which bytes will be uninitialised metadata when operating on a referenced value. The extra information is also needed to allow conversion to types representing numeric values, which will need somewhere to inherit width and range information from. The modification to `AllocRange` is shown below.

```
pub struct AllocRange {
    pub start: Size,
    // Replacing: pub size: Size,
    pub range: Option<Size>,
    pub width: Size,
}
```

compiler/rustc_middle/src/mir/interpret/allocation.rs:75

Single numeric values are passed around as the types `Scalar` and `ScalarInt`. Because pointers are in some cases stored using these types, they must also carry width and range information. These changes are shown below.

```
pub enum Scalar<Tag = AllocId> {
    Int(ScalarInt),
    // Replacing: Ptr(Pointer<Tag>, u8),
    Ptr(Pointer<Tag>, u8, u8),
}
```

compiler/rustc_middle/src/mir/interpret/value.rs:124

```
pub struct ScalarInt {
    data: u128,
    // Replacing: size: u8,
    range: u8,
    width: u8,
}
```

compiler/rustc_middle/src/ty/consts/int.rs:122

The changes to the compiler to propagate and update the extra information on these three types are fairly simple, but very widespread, including changes to object layout, constant evaluation, and vtable construction.

3.4 Pointer code generation

Code generation for atomic pointers makes some unsound assumptions about pointers for the Morello platform. To work around the limitations of pointer operations on some targets, Rust generates code which casts the pointer to an integer – this is not permissible on Morello and yields capability faults at runtime. Thankfully, the fix here is simple: we have added an option to the target settings to disable this cast on the Morello target, which is shown in line 12 of Figure 2. The new option is then checked in the code generation pass, as shown below. On Morello, which supports 128-bit atomics, this avoids down-casting pointers to a pair of `isize` (64-bit) integers.


```
// Replacing: if ty.is_unsafe_ptr() {
if ty.is_unsafe_ptr() && bx.target_spec().atomic_pointers_via_integers {
    let ptr_llty = bx.type_ptr_to(bx.type_isize());
    ptr = bx.pointerCast(ptr, ptr_llty);
    val = bx.ptrtoint(val, bx.type_isize());
}
compiler/rustc_codegen_ssa/src/mir/intrinsic.rs:471
```

3.5 Tweaks to LLVM

The port of LLVM for Morello [20] is mature, and we have encountered few bugs.

Some LLVM optimisations currently cause incorrect code generation when compiling Rust for Morello, so we have disabled them until they can be debugged. The optimisations currently disabled are function merging, visible in line 13 of Figure 2; and the Scalar Replacement Of Aggregates optimisation, which requires minor changes inside LLVM.

We also encountered an edge case in LLVM code generation that caused 6 byte structures to be emitted as 96 bit integers, which then triggered a miscompilation. This ultimately lead to spurious capability faults during execution of affected parts of programs. While the underlying bug has now been fixed in upstream Morello LLVM [6], we found it could be worked around by padding affected structures to larger sizes.

The remaining adjustments that were needed were to the Rust standard library.

3.6 MPSC

Rust includes a large suite of tests for the compiler and libraries. Running the standard library tests on Morello has been our primary means of detecting code generation bugs and standard library compatibility issues. We had initially anticipated that the standard library would need many changes, given its size, low level nature, and need for performance. In actuality, we have so far only needed to make minor changes, and the modifications to the MPSC component are by far the most involved.

The Multi-Producer Single Consumer primitive in Rust (MPSC) is part of the standard library’s concurrency module, `std::sync`. It provides communication channel types that can pass objects between threads. The implementation of MPSC demonstrates exactly the sort of problem one might expect to see on Morello: pointers are passed between threads, but converted to integers and back as part of the trip. The same storage is also used to hold non-pointer signalling values. The problem stems from the code below which directly converts a `usize` into a pointer type (the type of `inner`).

This isn’t compatible with our changes to the compiler because the `usize` used by MPSC can no longer be used to carry a valid pointer, and the `usize` type is no longer the same size as a pointer. This causes casting to fail during compilation, but would still cause run time errors if it *did* compile. In CHERI C, one might use `uintptr_t`, but as no equivalent type is currently defined in Rust we have simply replaced the integer types with pointers. MPSC has no need to perform complex arithmetic or any other integer-specific operations, so this doesn’t create any problems. The signalling values can simply be cast into pointers before use, and because they should never be dereferenced it doesn’t matter that the resulting capabilities are invalid.

```
#[inline]
pub unsafe fn cast_to_ptr(self) -> *mut () {
```

39:14 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

```
// Replacing: pub unsafe fn cast_to_usize(self) -> usize {
    mem::transmute(self.inner)
}

#[inline]
pub unsafe fn cast_from_ptr(signal_ptr: *mut ()) -> SignalToken {
// Replacing: pub unsafe fn cast_from_usize(signal_ptr: usize) -> SignalToken {
    SignalToken { inner: mem::transmute(signal_ptr) }
}

library/std/src/sync/mpsc/blocking.rs:53
```

It is interesting to note that since our changes to MPSC for the port to Morello, very similar changes have happened upstream. The upstream changes are the product of work on pointer provenance in the MIR interpreter project (MIRI) [17].

3.7 FFI types

Rust's standard library makes use of the C standard library via a wrapper called `libc`. All use of C APIs from Rust requires a wrapper or *Foreign Function Interface*. In testing on Morello we found a number of incompatible type definitions, where the original API expected a pointer or pointer sized value, and the Rust wrapper declared the value to be `usize`. This is easy to fix by simply replacing the types appropriately. An example is drawn below, where we have replaced integer types with Rust pointer types.

```
pub type off_t = i64;
pub type useconds_t = u32;
pub type blkcnt_t = i64;
pub type socklen_t = u32;
pub type sa_family_t = u8;
// Replacing: pub type pthread_t = ::uintptr_t;
pub type pthread_t = *mut PThread;
pub type nfds_t = ::c_uint;
pub type regoff_t = off_t;

#[allow(missing_copy_implementations)]
pub struct PThread { _opaque: [u8; 0] }

library/libc-0.2.93/src/unix/bsd/mod.rs:1
```

There are likely to be interfaces not covered by tests that will require further work, we have taken a conservative approach to all the changes we have made in the compiler making the minimal alterations to get correct compilation of the tests we have. A different approach might be sensible here, where all `ptr_t` types should natively be Rust pointer types for Morello targets. It is not clear how significant the knock-on effects of this change would be on external libraries which expect integer semantics for C pointer types.

```
pub type size_t = usize;
pub type ptrdiff_t = isize;
pub type intptr_t = isize;
// TODO: Perhaps on Morello this should be `* ::c_void`
pub type uintptr_t = usize;
pub type ssize_t = isize;
```

```
pub type pid_t = i32;
pub type uid_t = u32;
pub type gid_t = u32;
pub type in_addr_t = u32;
pub type in_port_t = u16;
// Replacing: pub type sighandler_t = ::size_t;
pub type sighandler_t = *const ();
pub type cc_t = ::c_uchar;

library/libc-0.2.93/src/unix/mod.rs:19
```

4 Performance analysis methodology

We outline our method for measuring Rust programs running on the prototype Morello platform.

4.1 Test hardware

We are using the Morello Prototype hardware [2]. The Morello CPU is a 7nm quad-core “Neoverse N1” based Armv8-A processor clocked at 2.5 GHz, connected to 16 GiB of 2933 MT/s DDR4 memory. The firmware is updated to Release 1.3 [2]. The prototype is packaged as an ATX-style motherboard in a standard ATX computer case.

4.2 Operating system

As described earlier, we use the port of FreeBSD to Morello, called CHERI BSD [34]. CHERI BSD is very stripped back, with minimal system utilities running for network connectivity and multi-user support, making it ideal as a benchmarking platform. The operating system was compiled with the CTSRD port of LLVM for Morello [20] using the `cheribuild.py` utility [9].

4.3 Disabling bounds checking

Rust provides automatic checking of array accesses, ensuring that subscripting will be in-bounds. This covers a subset of the bounds checking provided by Morello, which checks the bounds of all pointer accesses. To compare Morello hardware bounds checking to the software bounds checking emitted by `rustc`, we added a code generation flag to the compiler which disables software bounds checking: `-C drop_bounds_checks`. This has the effect of making all array accesses the same as Rust’s `unsafe fn slice::get_unsafe()`. Enabling this option on normal hardware makes the compiler unsound, while on Purecap Morello soundness is mostly, though not entirely, restored by hardware bounds checking. Bounds checks on Morello are precise up to 4 KiB blocks of memory, becoming imprecise beyond that as a result of the floating point representation used to store bounds information. For the sake of the performance comparisons made in this paper, we think that this approach is reasonable to give some indication of the cost of bounds checks relative to the cost of CHERI extensions.

This method of disabling software bounds checking is similar to previous work on measuring the runtime cost of Rust’s safety checks by Zhang et al., although not identical [42].

39:16 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

```
fn bounds_check(
    &mut self,
    block: BasicBlock,
    slice: PlaceBuilder<'tcx>,
    index: Local,
    expr_span: Span,
    source_info: SourceInfo,
) -> BasicBlock {
    // We do return an un-modified access when the -C drop_bounds_check
    // flag is enabled
    let gcx = *self.tcx;
    if gcx.sess.opts.cg.drop_bounds_checks {
        return block
    }

    // Otherwise, generate MIR code to check the bounds of an access
    /* ... */
}
```

compiler/rustc_mir_build/src/build/expr/as_place.rs:666

Our modification adds an early return to the code generation of bounds checking assertions, which would normally be inserted when array subscripting. In Zhang's work they modify a later stage of code generation to disable the lowering of these assertions to LLVM IR, as well as assertions that check for integer-overflow. We ran benchmarks with both approaches and found similar results, but we present the results for tests with the modification listed above only, as we are not investigating the cost of integer-overflow checks on Morello.

4.4 cargo bench

We ported the standard Rust benchmarking infrastructure to Morello. Programs are cross-compiled on a normal Apple M1 or x86_64 system. Rust's infrastructure includes a remote test harness which sends binaries to a remote system under test using network sockets. We built a port of the receiving part of this software, called `remote-test-server` in C, which runs on the Morello prototype. We can then use the standard `cargo bench` command to orchestrate building and running tests, which runs benchmarks repeatedly and reports a time per iteration in nanoseconds, and the variance between runs in \pm nanoseconds. For each benchmark in the suite we produce four results to complete a comparison matrix by varying two parameters. The first parameter is the hardware mode which can be one of two values: Hybrid or Purecap, as described in §2.3.1, this is varied using the `--target` option and can be either `aarch64-unknown-freebsd` for Hybrid, or `aarch64-unknown-freebsd-purecap` for Purecap mode. The second parameter is the bounds checking mode, using our `-C drop_bounds_checks` compiler flag, which can be either: bounds checking disabled (Rust_{DBC}), or enabled (Rust).

For example the results below are produced by a test from the crate `hashbrown`. Time is in nanoseconds (ns).

<code>hashbrown-0.11.2/clone_from_large</code>				
	Rust		Rust _{DBC}	
	Time/iter	±	Time/iter	±
Purecap	15,779	8	15,818	59
Hybrid	15,557	53	15,601	16

Before each run in the test matrix, Rust and the Rust standard library is also recompiled with flags to enable/disable software bounds checking appropriately for Rust/Rust_{DBC}.

From this data, we calculate Relative Error (R_E) for each test using the formula below, and count the variance into several bins in Table 1.

$$R_E = \frac{\pm \text{time/iter (ns)}}{\text{Mean benchmark time/iter (ns)}}$$

For example, the R_E of Hybrid with bounds checking for `hashbrown-0.11.2/clone_from_large` is 0.34%. The benchmarks with high relative error are tests which are very fast running, and are limited by the measurement precision of `cargo bench` in nanoseconds.

4.5 Line counts

We include a count of the number of lines of code with the table of benchmarked projects in Appendix A. This is intended as a rough guide to the scale of the benchmarks. The table also includes a count of lines of `unsafe` code, as an indication of how much Morello’s safety guarantees might add to the soundness of the specimen code. Both counts are gathered using the `cargo count` tool [19]. In total across the 19 projects, there are 108k lines of Rust source of which 1041 are unsafe.

The lines of code count gives the number of non-blank, non-comment lines of Rust source code in the repository of each project, for every project we use a benchmark suite from. The lines of `unsafe` count gives the number of non-blank, non-comment lines inside `unsafe` blocks and `unsafe` functions.

It should be noted that both of these measures are approximate and meant as a guide only. The number of lines of code is given for the whole of each project repository, and will include the library code under test, the benchmark suite, and any additional tests and Rust build scripts present, but will exclude any code in dependencies (of which each project generally has several). The number of lines of `unsafe` is particularly approximate, for two reasons. Firstly, the effects of `unsafe` are not well quantified by counting lines of code; the impact of a single line in a frequently and widely called function may be much higher than a large `unsafe` block that executes only once during the lifetime of the program. Further, a single line of `unsafe` can call an arbitrary amount of external unsafe code through the foreign function interface. Secondly, there are some known edge cases in the counting tool that may cause slightly inaccurate counts [19].

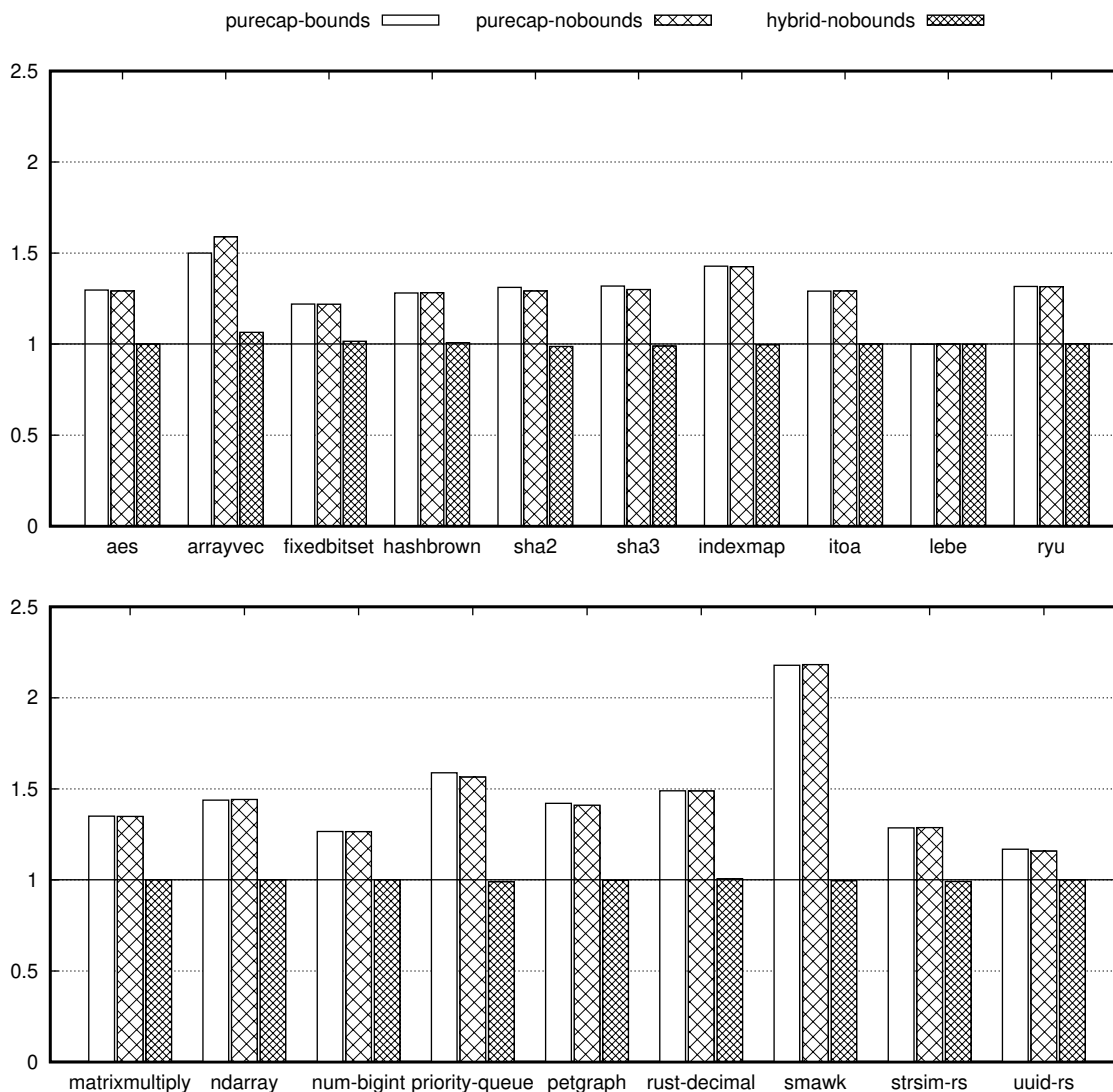
4.6 Test suites

A full list of test suites is available in Appendix A. These were picked for having low-level implementations of fast data structures, and having a small set of external dependencies. The suites cover arithmetic, array computations, cryptography, data-structures, Fourier analysis, hashing, and graph algorithms. Each test was taken directly from the standard Rust package repository (<https://crates.io>) and checked out to a version which is compatible with our Rust compiler. Cargo will satisfy dependencies with the latest available package which meets the requirements, but frustratingly that doesn’t include the Rust edition, so

39:18 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

for several packages we had to pin dependencies at older compatible versions too. Our test script automatically collects repositories from Git, and applies the minor patches to the `Config.toml` to pin dependencies where necessary. In total there is a little over 108k lines of Rust source code in these suites, and plenty more in the dependencies. There are 870 individual benchmarks which are run for each of the four modes described above.

5 Results



■ **Figure 3** Performance analysis of Rust programs in each of the modes described in §4, normalised to Hybrid-mode.

We present the aggregate execution times of each mode of Rust on Morello in the table below. They are normalised to Hybrid mode with software bounds checking enabled. Lower numbers indicate higher average performance.

■ **Table 1** Count of benchmark relative error into categories.

Relative error	Count
0 – 1%	3300
1 – 5%	142
≥ 5%	38

	Rust	Rust _{DBC}
Purecap	1.39	1.38
Hybrid	1.00	0.99

Individual benchmark suite results are shown in Figure 3. Again, for each of our benchmark results, we normalise against Hybrid-mode Rust with the Rust compiler’s software bounds checks enabled. We then take the geometric mean of each mode to aggregate the normalised performance change of the benchmarks within a benchmark suite, *i.e.* **hashbrown**, which internally contains 41 benchmarks. Bars above 1.0 are slower than Hybrid Rust with software bounds checks enabled, bars below are faster. These numbers represent the change in performance versus Rust on a modern Aarch64 machine today. Rust on Purecap Morello is approximately 39% slower than Rust on the equivalent Aarch64 machine.

In this section we will contrast the cost of bounds checking on the software and hardware, and consider the types of workload whose performance is affected the most by hardware bounds checking.

5.1 The cost of software bounds checking

Toggling software bounds checks (Rust_{DBC} vs. Rust) makes minimal difference in the performance we observe in our benchmark suites. The performance is similar because compiler optimisations remove most unnecessary bounds checks from code before runtime, and the performance cost of what remains is extremely low. Note that we are dropping bounds checks as aggressively as possible, even dropping them where Morello would provide imprecise bounds (as discussed in §4.3) – so we have here an upper-bound on the performance gain achievable in a sound implementation of Rust. We therefore conclude that re-engineering the Rust compiler to drop bounds checks on Morello is without merit.

5.2 The cost of hardware bounds checking

By-and-large the cost of hardware bounds checking is very significant, with only **lebe** showing a negligible difference. The performance hit, though large, is still relatively small compared to other techniques for always-on bounds checking for arbitrary binaries – such as running a program under **valgrind** or **purify** [13, 10]. For many applications this 39% cost for running on Morello is acceptable, and will likely only improve with any future hardware designs. This does open an interesting point for future work: off-loading as much bounds-checking to software, ideally statically enforced, and maintaining hardware bounds-checks where these guarantees cannot be enforced by software alone, could provide performance *and* strong safety guarantees.

5.3 Benchmarks

We present the results aggregated by benchmark suite in Figure 3. This shows consistent slowdowns in Purecap modes, but reveals some variation depending on the workload. The worst slowdowns are in `arrayvec` and `smawk` which perform a substantial number of array accesses, maximally exercising the capability protections afforded by Purecap Morello. `lebe`, which does endianness conversions for integers, is unaffected by differences under Purecap Morello, or whether bounds checks are enabled or not. The story is similar for other arithmetic heavy crates, like `fixedbitset`; code that leans less heavily on memory access suffers the lowest performance hit when running on Purecap Morello.

5.4 Validity of results

We address potential concerns for the validity of our results, and how we believe we have mitigated these.

5.4.1 Prototype hardware

The Morello prototype is just that, a prototype. There are known issues with real-world performance characteristics of the first prototype when operating in Purecap mode, which are reflected in our performance analysis. When the details of these limitations are released, or when a future revision of the hardware is released (to which ARM have *not* committed themselves), re-running our performance analysis to see if the overhead of capabilities can be brought in-line with normal performance of Aarch64 would be of significant interest. The comparison to Hybrid-bounds in this section gives the clearest picture of the behaviour of a comparable Aarch64 machine with no hardware bounds checking: in Hybrid mode the Morello prototype allows regular loads and stores using 64-bit pointers. That being said, the implementation is still customised compared to normal Aarch64 and micro-architectural quirks might be present which mean this comparison is invalid compared to “wild” Aarch64 implementations. Table 1 shows that our measurements are precise and run-to-run variance is very small in the vast majority of benchmarks.

5.4.2 Choice of benchmark suite

We have used benchmark suites included with large popular projects from the Rust package repository `crates.io`, spanning a range of different applications: hashing and cryptography (arithmetic heavy); tree and graph like data structures (indirection heavy); and big integer and matrix operations (array heavy). This is different to previous approaches to measuring Rust runtime performance, notably in recent work by Zhang et al., where synthetic micro-benchmarks were used [42]. We have chosen to use benchmarks that test real world Rust code as this is more likely to give a representative picture of how the cost of bounds checking will be felt in regular Rust programs. We found some benchmarks (for example, `RustFFT` and `itertools`) would compile for Morello, but not run correctly on Purecap modes because of assumptions about pointers which are invalid in Purecap mode.

6 Related Work

In this section we discuss the context for Morello and the necessary related work that enabled us to build the Rust for Morello compiler.

6.1 Rust and type-safe systems programming

Rust itself has been a subject of significant academic research. On the topic of formally specifying Rust, notable work includes Rust Belt [16] and Oxide [38]. We are interested in the formal treatment of Rust, especially in the context of the formal specification of Morello, but for this paper we consider this to be future work.

Additionally, there has been a history of other attempts at designing safe systems programming languages which, like Rust, use some notion of lifetimes to manage memory allocation and prevent memory errors. Cyclone [14] is one such language, which uses type-level *regions* among other type-system machinery to ensure safety while still admitting low-level systems code. The specifics of preventing memory errors in safe systems programming languages is discussed more in §6.3.

6.2 Prior work porting Rust to CHERI

There is existing work that explores extending the Rust compiler to target CHERI MIPS hardware. Nicholas Sim’s MSc thesis [32] describes the initial steps of targeting CHERI in the Rust compiler. Crucially, Sim chose the 128-bit representation of `usize`, whereas we opted for 64-bit `usize` (discussed in §3.1). This has cascading effects on the rest of the compiler and leads to various issues.

A major concern of Sim was divergence from upstream, which we agree is a real consideration. The wording of the documentation does not make it totally clear how large `usize` should be on Morello, it states [29]:

`usize`

The pointer-sized unsigned integer type.

The size of this primitive is how many bytes it takes to reference any location in memory. For example, on a 32 bit target, this is 4 bytes and on a 64 bit target, this is 8 bytes.

There is clearly room for argument one way or the other. We are of the opinion that 64-bit `usize` on Morello is compatible with the spirit of this definition, Morello is a 64-bit platform with 64-bit addresses, so in order to range over all memory addresses it suffices for `usize` to be 64-bit. Conversely, supporting Sim’s initial choice, to reference a location in memory on Morello it is necessary to use a capability which is 128-bit.

Sim also reported a number of technical issues resulting from 128-bit `usize`, including performance problems, and the need for inserting integer-truncate and integer-extend operations when calling LLVM intrinsics like `memcpy` and `inttoptr`. We have not had to contend with these issues in our implementation, and agree with Sim’s assessment that 64-bit `usize` is preferable for this reason.

6.3 Bounds checking

There has been significant prior research on enforcing memory safety by preventing out-of-bounds accesses, either by statically proving accesses will be in-bounds, or by augmenting programs with dynamic bounds checking (or a combination of both).

A key result on statically eliminating bounds violations comes from Xi and Pfenning, who demonstrate that, with a *dependent type system*, array accesses may be accompanied by *proofs* that the computed index is within the array bounds [40]. This gives a compile-time guarantee that no bounds violations will occur on any array access, and thus safety can be maintained without the need for any dynamic checks. This was done in the context of a high-level, ML-like language, so there was no need to deal with gnarly C-like pointer arithmetic.

Prior work also explores static elimination of *some* bounds checking as a compiler optimisation. An optimising compiler or just-in-time compiler may seek to reduce the number of required bounds checks in a program by proving that some subset of accesses will always succeed. Early work demonstrated how this may be done with dataflow analysis [12]. This has been of particular interest to implementors working with the Java programming language, as the specification states that out-of-bounds array accesses must be caught at runtime, but bounds checks are not expressible in standard Java bytecode. To address this problem, lightweight techniques for eliminating some bounds checks at runtime within a Just-in-Time compiler have been developed [7].

In addition to minimising or eliminating bounds checks when compiling high-level languages like Java, there has been interest in enforcing memory safety in existing C and C++ programs. Doing automatic bounds checking in C or C++ is difficult because of the need to track, at run-time, what object each pointer value is intended to point to [15]. Unlike languages like Java, C and C++ allow programmers to do *pointer arithmetic* in order to compute (for example) an index into the middle of an array. These pointers into the interiors of objects may be written to a data structure or passed to a function within the application; in order to check the latter dereferencing of such pointers, it is necessary to keep track of the *intended referent* of the pointer as the pointer value flows through the program. There have been several approaches to solving this problem and preventing or catching out-of-bounds errors in existing systems code.

One heavyweight approach is *binary instrumentation*. Binary instrumentation tools like `purify` and `valgrind` are able to arbitrarily add metadata to pointers and detect a wide range of memory referencing errors [13]. Of course, as these tools are designed for debugging, they impose a significant runtime overhead that makes them impractical for use in production software. Additionally, `purify` can end up missing some memory safety violations if pointer arithmetic happens to yield a pointer to a different but still valid object.

Another approach involves changing pointer representations. Systems like SafeC [5] and Cyclone [14] use an extended pointer representation (“fat pointers”) to record information about the intended referent. These fat pointers work similarly to Morello’s capabilities, but their enforcement mechanisms are implemented in software rather than in hardware. SafeC and Cyclone allow for dynamic checking, but also produce code that is incompatible with external, unchecked code. A different approach, which maintains backwards compatibility with legacy C code, was proposed by Jones and Kelly: store the address ranges of live objects and ensure that pointer arithmetic never crosses out of the one object and into another valid object [15]. In this approach, address ranges are stored in a global table, and the table is referenced before every pointer arithmetic operation. Unsurprisingly, this introduces a large amount of overhead at run-time – over 5x overhead on many programs and, in subsequent work that extended this approach to cover a larger class of C programs, over 11x overhead [30]. Later work by Dhurjati and Adve demonstrated that the overhead of backwards-compatible array bounds checks in C could be drastically reduced by exploiting a fine-grain partitioning of memory called Automatic Pool Allocation [10].

7 Future work

The relationship between Rust and Morello seems clear, both aim to provide a programmer with guarantees that memory safety violations cannot happen in their programs. Rust achieves this through linear types, and runtime bounds checks, and Morello achieves this

through hardware pointer provenance and hardware bounds checking. It would be interesting to show a formal relationship between the semantics of Rust and the guarantees provided by Morello.

With the Morello prototype running in Hybrid mode it is possible to mix the use of capability instructions, where the hardware enforces bounds checking at run time, with normal load/store instructions. Through static analysis it might be possible to find the pointers in Rust which require runtime bounds checks and to promote those to capabilities on a per-pointer basis: thereby pushing bounds checks into hardware, rather than by emitting additional code. This could also be used to martial code in/out of unsafe components or through the FFI. Using capabilities to compartmentalise each `unsafe` block and each FFI call could bring new safety guarantees to Rust code which interacts with components that could introduce memory safety violations.

If ARM later commits to incorporating CHERI extensions into the ARM ISA at large, then incorporating the changes from our prototype Rust compiler into upstream Rust would be a natural extension of this work.

8 Conclusions

Morello provides a costly, but not impractical, means for achieving always-on memory safety. Like Rust, Morello has been built with the benefit of hindsight: memory safety is the most significant problem for building bug-free code. We have found that the overhead for Morello in this first prototype is around 39%, and elimination of Rust's bounds checks might yield a 1% speed up.

There is a two-way benefit for a programmer using Rust on Morello. When using safe Rust, a programmer knows that for all their safe code they cannot get a CHERI protection error which would cause their code to fault at runtime – unlike if they were to program in C. When using `unsafe` Rust, the programmer knows that if things do go terribly wrong, the Morello platform will protect them from memory errors which could cause security vulnerabilities.

The Rust for Morello compiler presented in this paper is fully featured. We have demonstrated the compiler works for a significant chunk of wild Rust code, without modification. In-all, the code we've compiled and run for Morello is around 108k lines of Rust, all from the Rust Crates repository, and all without modification to the Rust code.

This compiler forms the groundwork for future research into safe systems programming on hardware designed from the ground up to provide memory safety.

References

- 1 Arm. *Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture*. Arm, 2020.
- 2 ARM. Morello project – release notes, January 2022. last accessed: July 25, 2022. URL: <https://git.morello-project.org/morello/docs/-/blob/morello/mainline/release-notes.rst>.
- 3 ARM and contributors. The android/morello release, April 2022. last accessed: September 28, 2022. URL: <https://git.morello-project.org/morello/docs/-/blob/morello/mainline/android-readme.rst>.
- 4 ARM and contributors. Morello project – linux, August 2022. last accessed: September 28, 2022. URL: <https://git.morello-project.org/morello/kernel/linux>.

- 5 Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/178243.178446.
- 6 Silviu Baranga. Don't replace a 96-bit memcopy with a capability load/store, September 2022. URL: https://git.morello-project.org/morello/llvm-project/-/merge_requests/205.
- 7 Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: Eliminating array bounds checks on demand. *SIGPLAN Not.*, 35(5):321–333, May 2000. doi:10.1145/358438.349342.
- 8 Common Weakness Enumeration. 2022 CWE Top 25 Most Dangerous Software Weaknesses. Technical report, MITRE, August 2022. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.
- 9 CTSRD CHERI. cheribuild, 2022. last accessed: July 25, 2022. URL: <https://github.com/CTSRD-CHERI/cheribuild>.
- 10 Dinakar Dhurjati and Vikram Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 2006 International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 2006. URL: <http://llvm.org/pubs/2006-05-24-SAFECODE-BoundsCheck.html>.
- 11 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/512529.512563.
- 12 Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst.*, 2(1–4):135–150, March 1993. doi:10.1145/176454.176507.
- 13 Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.
- 14 Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1029873.1029883.
- 15 Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the Third International Workshop on Automated Debugging, AADEBUG 1997*, 1997.
- 16 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), 2017. doi:10.1145/3158154.
- 17 Ben Kimock. Remove ptr-int transmute in std::sync::mpsc, April 2022. URL: <https://github.com/rust-lang/rust/commit/dec73f5>.
- 18 Steve Klabnik, Carol Nichols, et al. *The Rust Programming Language*. The Rust Project Developers, 2021. URL: <https://doc.rust-lang.org/1.55.0/book/>.
- 19 Kevin Knapp. cargo-count, November 2017. URL: <https://github.com/kbknapp/cargo-count>.
- 20 LLVM Project and CTSRD CHERI. CTSRD llvm-project, 2022. last accessed: July 25, 2022. URL: <https://github.com/CTSRD-CHERI/llvm-project>.
- 21 Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, October 2014. doi:10.1145/2692956.2663188.
- 22 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring c semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290380.
- 23 Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. last accessed: July 25, 2022. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

- 24 Miguel Ojeda. [PATCH 00/13] [RFC] Rust support, April 2021. URL: <https://lore.kernel.org/lkml/20210414184604.23473-1-ojeda@kernel.org/>.
- 25 Rust project contributors. [Pre-RFC] usize is not size_t, September 2021. URL: <https://internals.rust-lang.org/t/pre-rfc-usize-is-not-size-t/15369>.
- 26 Rust project contributors. The Rust Standard Library - Primitive Type usize, 2021. URL: <https://doc.rust-lang.org/1.55.0/std/primitive.usize.html>.
- 27 Rust project developers. Rust 0.1, 2012. URL: <https://github.com/rust-lang/rust/releases/tag/0.1>.
- 28 Rust project developers. *The Rust Reference*. The Rust Project Developers, 2021. URL: <https://doc.rust-lang.org/1.55.0/reference/>.
- 29 Rust project developers. *usize - Rust*. The Rust Project Developers, 2021. URL: <https://doc.rust-lang.org/std/primitive.usize.html>.
- 30 Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.
- 31 Nicholas Sim. Support index size != pointer width. <https://github.com/rust-lang/rust/issues/65473>, October 2019. last accessed: November 28, 2022.
- 32 Nicholas Sim. Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language. Master’s thesis, University of Cambridge, August 2020. URL: <https://nw0.github.io/cheri-rust.pdf>.
- 33 The Chromium Projects. Memory Safety. <https://www.chromium.org/Home/chromium-security/memory-safety/>. last accessed: July 25, 2022. URL: <https://www.chromium.org/Home/chromium-security/memory-safety/>.
- 34 The FreeBSD Project and CTSRD CHERI. CTSRD-CHERI cheribsd, 2022. last accessed: March 4, 2021. URL: <https://github.com/CTSRD-CHERI/cheribsd>.
- 35 Aaron Turon. Rust blog: Abstraction without overhead: Traits in rust, May 2015. URL: <https://blog.rust-lang.org/2015/05/11/traits.html>.
- 36 Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019. doi:10.48456/tr-941.
- 37 Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. doi:10.48456/tr-951.
- 38 Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, 2019. arXiv:1903.00982.
- 39 Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, May 2020. doi:10.1109/SP40000.2020.00098.
- 40 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI ’98*, pages 249–257, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/277650.277732.

- 41 Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '22*, pages 545–557, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3352460.3358288.
- 42 Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22*, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3551349.3559494.

A Test suite

Benchmark	Version	Description	SLOC	Unsafe
<code>arrayvec</code>	0.7.2	A vector with fixed capacity, backed by an array.	2,059	26
<code>block-ciphers/aes</code>	0.7.2	Pure Rust implementation of the Advanced Encryption Standard.	4,218	25
<code>fixedbitset</code>	0.3.1	A simple bitset collection.	1,431	4
<code>hashbrown[†]</code>	0.11.2	Google's SwissTable hash map.	8,454	182
<code>hashes/sha2</code>	0.10.2	Pure Rust implementation of the SHA-2 hash function family.	1,086	2
<code>hashes/sha3</code>	0.10.1	SHA-3 (Keccak) hash function	320	0
<code>indexmap[†]</code>	1.0.0	A hash table with consistent order and fast iteration.	6,092	3
<code>itoa</code>	1.0.3	Fast integer primitive to string conversion.	324	5
<code>lebe</code>	0.5.0	Tiny, dead simple, high performance endianness conversions with a generic API.	527	40
<code>matrixmultiply</code>	0.3.2	General matrix multiplication for f32 and f64 matrices.	3,898	22
<code>ndarray</code>	0.15.6	An n-dimensional array for general elements and for numerics.	25,508	340
<code>num-bigint</code>	0.4.3	Big integer implementation for Rust.	12,541	0
<code>petgraph[†]</code>	0.6.0	Graph data structure library.	19,559	5
<code>priority-queue[†]</code>	1.3.1	A Priority Queue implemented as a heap with a function to efficiently change the priority of an item.	3472	68
<code>rust-decimal</code>	1.23.1	Decimal number implementation written in pure Rust suitable for financial and fixed-precision calculations.	11,469	0
<code>ryu</code>	1.0.12	Fast floating point to string conversion.	2,930	317
<code>smawk</code>	0.2.0	Functions for finding row-minima in a totally monotone matrix.	740	0
<code>strsim</code>	0.10.0	Implementations of string similarity metrics.	837	0
<code>uuid-rs</code>	1.3.0	A library to generate and parse UUIDs.	3,505	2
Total			108,970	1,041

Tests marked with [†]required patches to their Config.toml to pin compatible versions of dependencies.

On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage

João Mota  

NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal

Marco Giunti 

NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal
School of Computing, Engineering & Digital Technologies, Teesside University, Middlesbrough, UK

António Ravara 

NOVA LINCS and NOVA School of Science and Technology, Caparica, Portugal

Abstract

Typestates are a notion of behavioral types that describe protocols for stateful objects, specifying the available methods for each state. Ensuring methods are called in the correct order (protocol compliance), and that, if and when the program terminates, all objects are in the final state (protocol completion) is crucial to write better and safer programs. Objects of this kind are commonly shared among different clients or stored in collections, which may also be shared. However, statically checking protocol compliance and completion when objects are shared is challenging. To evaluate the support given by state of the art verification tools in checking the correct use of shared objects with protocol, we present a survey on four tools for Java: VeriFast, VerCors, Plural, and KeY. We describe the implementation of a file reader, linked-list, and iterator, check for each tool its ability to statically guarantee protocol compliance and completion, even when objects are shared in collections, and evaluate the programmer's effort in making the code acceptable to these tools. With this study, we motivate the need for lightweight methods to verify the presented kinds of programs.

2012 ACM Subject Classification Theory of computation → Program reasoning; Theory of computation → Logic and verification; Theory of computation → Separation logic

Keywords and phrases Java, Typestates, VeriFast, VerCors, Plural, KeY

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.40

Category Experience Paper

Related Version *Extended Version*: <https://arxiv.org/abs/2209.05136>

Supplementary Material

Software: <https://github.com/jdmota/tools-examples/tree/ecoop-2023>
archived at [swh:1:dir:9b9f9f7a269c44c04c57d5f66ff27884de02d4bc](https://swh.1.dir:9b9f9f7a269c44c04c57d5f66ff27884de02d4bc)

Funding Partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI) and NOVA LINCS (UIDB/04516/2020).

João Mota: Partially supported by FCT.IP (2021.05297.BD).

Marco Giunti: Partially supported by Dstl, reference: ACC2028868.

Acknowledgements We would like to thank several members of the developer teams for the detailed responses and enlightening discussions, in particular, Bart Jacobs (VeriFast), Marieke Huisman (VerCors), Lukas Armbrorst (VerCors), Reiner Hähnle (KeY), Eduard Kamburjan (KeY), and Richard Bubel (KeY). Their feedback was indispensable for the development of this study.



© João Mota, Marco Giunti, and António Ravara;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 40; pp. 40:1–40:29

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In object-oriented programming, one naturally defines objects where their methods' availability depends on their internal state [49]. For example, the `next` method of an iterator can only be called if there are items to be retrieved, otherwise it throws an exception. One might represent their intended usage protocol with an automaton or a state machine [60, 61, 25]. **Behavioral types** [34, 3], when used for object-oriented languages, allow us to statically check if all code of a program respects the protocol of each object, helping us to write safer programs with fewer errors. The crucial properties verified are **protocol compliance**, ensuring methods are called in the correct order, and **protocol completion**, guaranteeing that if and when the program terminates, all the objects are in their final states, ensuring required method calls are not forgotten, and that resources are freed, for example, that we close all sockets. Bravetti et al. present a formal treatment of these properties [19].

In session types approaches, objects with protocols are usually forced to be used in a linear way to avoid race conditions, which reduces concurrency and restricts what a programmer can do [32, 34]. Given that **sharing of objects** is very common, it should be naturally supported, without putting too much burden on the programmer. For example, pointer-based data structures, like linked-lists, usually rely on internal sharing (i.e. aliasing). Such collections may also be used to store an arbitrary number of objects in different states which need to be tracked. Developing these data-types, and applications using them, is often challenging. To our knowledge, there is no typestate-oriented support for tracking the states of objects in collections, while ensuring protocol compliance and completion. Given this, the present study has the objective of answering the following research question (in Section 3.4)

RQ: *Are current static verification tools capable of verifying protocol compliance and completion even when objects are shared in collections?*

To study the contributions and limitations of the state of the art, we report our experience in verifying protocol compliance and completion with four tools for Java: VeriFast [39, 38], VerCors [33, 13], Plural [10], and KeY [1]. We picked these because of their rich features for verification, and because they are actively maintained (with the exception of Plural). We believe these cover the most used static analysis techniques which can be instructed to perform typestate verification. *VeriFast* checks programs annotated with method contracts written in **separation logic** [51, 55]. *VerCors*, however, uses **permission-based concurrent separation logic** to check programs, inspired by Chalice [44, 45]. *Plural* verifies that the protocols of objects are respected with **typestates** [59]. It introduces **access permissions** which combine typestate and object aliasing information, allowing state to be tracked and modified even when objects are shared, allowing for more uses beyond the “single writer vs multiple readers” model of fractional permissions [17]. *KeY* verifies sequential Java programs with specifications written in JML [43], based on first-order Java **dynamic logic** (JavaDL) [27]. It supports a great number of Java features and provides an interactive theorem prover with a high degree of automation and useful tactics to guide proofs. OpenJML is another verification tool based on JML [22]. Given that the differences [16] are not significant for our use case, and since KeY has an interactive prover, we focus our study on KeY.

To our knowledge, no such comparison study was previously done that focused on protocol compliance and completion. As we will observe, the running examples may look simple but constitute real challenges to these tools, especially since three of them are based on contracts, not behavioral types. Our conclusions support advocating for *lightweight verification methods* directed at these protocol related properties. The contributions of this paper are:

- Code **implementations** and **examples** of using file readers, linked-lists, and iterators (when possible), similar in all four tools, and **specifications** appropriate to each tool to verify the desired properties;
- An assessment if the tools can check **protocol compliance**, and if they can guarantee **protocol completion**, even with objects **shared in collections**;
- An evaluation of the **programmer’s effort** in making the code examples acceptable to each tool, justifying the need for lightweight methods to verify these kinds of programs.

With regards to our level of experience, we are knowledgeable in the concepts used, having applied them in different settings, and have had practical experience with VeriFast before conducting this study. Regarding the other tools, we report our experience in using them for the first time. To validate this study, our assessments were shared with the development teams. From the responses we got, they found the examples to be interesting and challenging. All the received feedback was very valuable in helping us to refine and confirm our conclusions.

This paper is structured as follows: Section 2 provides an overview of each tool; Section 3 presents **code implementations and specifications**, and if they were **accepted**, thus reporting our experience and answering the **RQ** (Section 3.4); Section 4 discusses our **assessments**; Section 5 discusses relevant work; finally, Section 6 presents our **conclusions**.

2 Background

2.1 VeriFast

VeriFast is a modular verifier for (subsets of) C and Java programs annotated with method contracts (pre- and post-conditions) written in **separation logic** [39, 38, 51, 55]. Besides the points-to assertions from separation logic, specifications support the definition of inductive data types, predicates, and fixpoint functions. Additionally, **deductive reasoning** [5] is supported via the definition of lemmas and the insertion of assertions in key program points to guide verification. Furthermore, it comes with an IDE allowing one to observe each step of a proof when an error is encountered. VeriFast is then able to statically check that contracts are respected during execution and that programs will not raise errors such as null pointer exceptions or perform incorrect actions such as accessing illegal memory. Nonetheless, VeriFast’s support for generics is still limited.¹

When declaring predicates, one may also specify **output parameters**. These appear after input parameters separated with a semicolon. Output parameters need to be precisely defined in the predicate definition and allow its user to “extract” them. In Listing 1, the parameter `b` is defined as an output parameter of the `account` predicate (line 1). It is precisely defined to be the value of the `balance` field of the account object. The `|->` symbol represents the points-to assertion while `&*&` represents the separating conjunction binary operator. This balance can then be “extracted” using the `?` symbol in a pre-condition (line 5), existentially quantifying the name `b`, which can then be mentioned in the post-condition (line 6), indicating the effect of a deposit given the current balance `b`.

■ **Listing 1** Output parameters example.

```

1 //@ predicate account(Account a; int b) = a.balance |-> b &*& b >= 0;
2 class Account {
3     int balance;
4     void deposit(int value)

```

¹ <https://github.com/verifast/verifast/issues/271>

40:4 On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage

```
5     //@ requires account(this, ?b) &*& 0 < value;
6     //@ ensures account(this, b + value);
7     {
8         balance += value;
9     }
10 }
```

For the sharing of memory locations, VeriFast has built-in support for **fractional permissions**, associating a number coefficient between 0 (exclusive) and 1 (inclusive) to each heap chunk [17]. By default, the coefficient is 1, which allows reads from and writes to a memory location. A number less than 1 allows only for reads. The programmer may provide coefficient patterns in the form of expressions, such as literal numbers or variables, or in the form of existentially quantified names (like *f* in line 2 of Listing 2). These patterns may be applied to points-to assertions but also to predicates. Applying a coefficient to a predicate is equivalent to multiplying it by each coefficient of each heap chunk mentioned in the predicate's body. Additionally, VeriFast supports the automatic splitting and merging of fractional permissions. Counting permissions are also supported via a trusted library [17].

■ Listing 2 Coefficient pattern example.

```
1 int getBalance()
2     //@ requires [?f]account(this, _);
3     //@ ensures [f]account(this, result);
4     { return balance; }
```

For C programs, VeriFast also supports leak checking: after consuming post-conditions, the heap must be empty. However, the programmer can leak certain resources with the `leak` command. For Java programs, leaking is always allowed.

2.2 VerCors

VerCors is a verifier for concurrent programs written in Java, C, OpenCL and PVL (Prototypal Verification Language), and annotated with method contracts [33, 13]. The specifications employ a logic based on **permission-based concurrent separation logic** [50]. The verification procedure is modular, checking each method in isolation given a contract with pre- and post-conditions. As in VeriFast, lemmas can be defined and assertions introduced to guide verification. Although there are other tools that perform static verification on annotated programs, VerCors focuses on supporting different concurrency patterns of high-level languages, and is designed to be language-independent. Support for inheritance and exceptions is still being worked upon based on theoretical work by Rubbens [56].² Internally, VerCors uses the Viper backend [48], which in turn uses Z3 [23].

VerCors supports two styles for specifying access to memory locations: **permission annotations**, following the approach of Chalice [44, 45]; and **points-to assertions** of separation logic, as in VeriFast. Both styles of specification have been shown to be equivalent by Parkinson and Summers [53]. The equivalence is presented in Figure 1. On the left-hand-side it is shown the use of a permission annotation, `Perm`, to request access to variable `var`, with fractional permission `p`, and storing a value equal to `val`. The `**` symbol represents the separating conjunction operator. On the right-hand-side it is shown the equivalent `PointsTo` assertion. Permission annotations are very useful because they allow us to refer to values in variables without the need to use new names for them.

² <https://vercors.ewi.utwente.nl/wiki/#inheritance-1>

```
Perm(var, p) ** var == val ≡ PointsTo(var, p, val)
```

■ **Figure 1** Permission annotations equivalent to points-to assertions.

As an example, List. 3 shows a method contract which requires exclusive permission to write in field `val` (line 2), and ensures that the new value is equal to the old one incremented by one (line 3).

■ **Listing 3** Permissions example.

```
1  /*@
2   requires Perm(val, 1);
3   ensures Perm(val, 1) ** val == \old(val) + 1;
4  */
5  void increment(){
6   val = val + 1;
7  }
```

VerCors also has support for **ghost code**, including ghost parameters and results in methods. These are declared in methods' contracts with the `given` and `yields` keywords, respectively. When calling a method, one uses the `with` and `then` keywords to assign ghost parameters and retrieve return values, respectively. These features are exemplified in Listing 4 where a `sum` method yields a ghost result given two ghost parameters, `x` and `y`. Line 13 shows how this method may be used. Ghost code is useful to keep track of intermediate results, which only exist for the purpose of verification.

■ **Listing 4** Ghost code example.

```
1  /*@
2   given int x;
3   given int y;
4   yields int res;
5   ensures res == x + y;
6  */
7  void sum() {
8   /*@ ghost res = x + y;
9  }
10
11 void main() {
12   sum() /*@ with {x=3; y=2;} then {result=res;} @*/;
13 }
14 }
```

2.3 Plural

Bierhoff and Aldrich addressed the problem of substitutability of subtypes while guaranteeing **behavioral subtyping** in a object-oriented language [9]. The specification technique models protocols using abstract states, incorporating *state refinements* (allowing the definition of substates, thus supporting substitutability of subtypes), *state dimensions* (which define orthogonal states corresponding to AND-states in Statecharts [28]), and *method refinements* (allowing methods in subclasses to accept more inputs and return more specific results). The approach is similar to pre- and post-condition based ones but provides better information hiding thanks to the **typestate** abstraction [59].

Bierhoff and Aldrich then built on previous work and developed a sound modular protocol checking approach, based on typestates, to ensure at compile-time that clients follow the usage protocols of objects even in the presence of aliasing [10]. For that, they developed the

40:6 On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage

notion of **access permissions** which combine typestate and object aliasing information. The approach was realized in Plural, a static verifier they developed for Java, as a plugin for Eclipse. As far as we know, not all Java's features are supported, such as exceptions. Although it is not maintained any longer, one can install it in Eclipse Juno (an old version from 2012) from its source. Given its support for rich access permissions, and direct application of the typestate abstraction, we believe its study is still very relevant.

An *access permission* tracks how a reference is allowed to read and/or modify the referenced object, how the object might be accessed through other references, and what is currently known about the object's typestate. To increase the precision of access permissions, Bierhoff and Aldrich introduced *weak permissions* (such as *share* and *pure*), where an object can be modified through other permissions. The proposed permissions include a *state guarantee* which ensures that an object remains in that state even in the face of interference. Additionally, they track *temporary state assumptions* which are discarded when they become outdated. All kinds of permissions are described in Table 1. A comprehensive survey on permission-based specifications was presented by Sadiq et al. [57].

■ **Table 1** Permissions in Plural.

Kind	Access to the referenced object	Access other aliases may have
Full	read and write	read-only
Pure	read-only	read and write
Immutable	read-only	read-only
Unique	read and write	none
Shared	read and write	read and write

In Listing 5 is an example of an iterator. In lines 1-3, it is stated that this object may be in two distinct states, **available** or **end**. These are defined as refinements, or subtypes, of the root state **alive**, a state common to all objects. Then it is specified that the **hasNext** method may be called in the **alive** state with just **pure** permission (line 5). If the method returns **true**, we further know that the iterator is in the **available** state, allowing us to refine our knowledge (line 6). Otherwise, the iterator is in the **end** state (line 7). The **next** method requires **full** permission and that the object be in the **available** state, and then it can only ensure it is in the **alive** state (line 10).

■ **Listing 5** Iterator example.

```
1 @Refine({
2   @States(value={"available", "end"}, refined="alive")
3 })
4 interface Iterator<E> {
5   @Pure("alive")
6   @TrueIndicates("available")
7   @FalseIndicates("end")
8   boolean hasNext();
9
10  @Full(requires="available", ensures="alive")
11  E next();
12 }
```

These kinds of permissions may be split to allow sharing of an object, and joined back together to allow one to potentially restore *unique* permission. **Fractional permissions** are used to track how much a permission was split [18]. Furthermore, different fractions can be mapped to different state guarantees through a *fraction function*, thus tracking for each state guarantee separately how many other permissions rely on it.

Beckman et al. extended the approach to verify the correctness of usage protocols in concurrent programs, statically preventing races on the abstract state of an object as well as preventing violations of state invariants [7]. This approach uses atomic blocks and was also realized in Plural. In this solution, access permissions are used as an approximation of the thread-sharedness of objects. For example, if *pure* or *share* permissions are used, it means that other references can modify the object, and it is assumed that this includes concurrent modifications. In this scenario, temporary state assumptions are discarded, unless the access is synchronized. Furthermore, accessing fields of an object with *share*, *pure*, or *full* permissions, must be performed inside atomic blocks.

Beckman later presented a similar approach which uses synchronization blocks instead of atomic blocks as the mutual exclusion primitive, given that the former are in more wide use [6]. Since programmers are required to synchronize on the receiver object, it becomes implicit to the analysis which parts of the memory are exclusively available, so programmers are not required to specify which parts of the memory are protected by which locks. However, this also implies that private objects cannot be used for the purposes of mutual exclusion.

2.4 KeY

KeY is a verifier for sequential Java programs [1]. Specifications are provided in Java comments in JML*, an extension of the Java Modeling Language (JML) [43]. JML is based on the design by contract paradigm with class invariants and method contracts [46]. Class invariants describe properties that must be preserved by all methods. Method contracts are composed by pre-conditions, post-conditions, and frame conditions, indicating the heap locations which a method may modify. With this, KeY can employ modular verification.

The example in Listing 6 uses a specification technique based on **dynamic frames** [40] and is adapted from the *Cell* example by Smans et al. [58]. The cell's specification starts by declaring a set of locations called `footprint` (line 4) composed only by the `x` field (line 6). The `accessible` annotation (line 5) specifies that the set `footprint` will only change if a location in the set mentioned on the right-side of the colon changes. In this case, `footprint` is constant. Then an invariant is stated (line 9) and it is specified that it only depends on locations in `footprint`. The `setX` method will not throw exceptions (indicated with the `normal_behavior` annotation in line 12), it assigns only to locations in the `footprint` (line 13), requires the parameter to be positive (line 14), and ensures the next call to `getX` will return the given value (line 15). Since it does not perform reads from heap locations, the `accessible` annotation is omitted.

■ Listing 6 JML specification example.

```

1  class Cell {
2      private int x;
3
4      /*@ model \locset footprint;
5         @ accessible footprint: \nothing;
6         @ represents footprint = x;
7         @*/
8
9      //@ invariant x > 0;
10     //@ accessible \inv: footprint;
11
12     /*@ normal_behavior
13        @ assignable footprint;
14        @ requires value > 0;
15        @ ensures getX() == value;
16        @*/

```

```

17 void setX(int value) {
18     x = value;
19 }
20 }

```

As far as we know, KeY is the verification tool that supports the greatest number of Java features among the other static verification tools for Java, allowing one to verify real programs considering the actual Java runtime semantics. This includes reasoning about inheritance, dynamic method lookup, runtime exceptions, and static initialization. A consequence of this is that, as the members of the KeY team point out, KeY is not overly suitable for the verification of algorithms that require abstracting away from the code since KeY's main goal is the verification of Java programs [20].

KeY's is based on an interactive theorem prover for first-order Java dynamic logic (JavaDL) [27], which can be seen as a generalization of Hoare logic [30]. An important part in the construction of proofs in KeY is symbolic execution. This process takes every possible execution branch and transforms the program leading to a set of constraints, which can then be verified against the specification. KeY provides a semi-automated environment where the user may choose to apply every step of the proof, apply a strategy macro, combining several deductive steps, or execute an automated proof search strategy. As well as offering a high degree of automation, KeY supports SMT solvers, such as Z3 [23], which are often useful to solve arithmetical problems [20]. More details on how to use KeY may be found online.³

The most common strategy macros, which we have significantly used in our experiments, are: propositional expansion (to apply propositional rules); finish symbolic execution (to apply only rules for modal operators of dynamic logic, thus executing Java programs symbolically); close provable goals (automatically close all goals that are provable, but do not apply any rules to goals which cannot be closed).

3 Experiments

In this section, we start by giving an overview of the examples that are going to be used, showing the object usage protocols and their implementations, as well as, client code using the presented objects (in Section 3.1). The code is in Java, a language supported by all the aforementioned tools, which given its object-oriented nature, is well-suited for building objects with protocol where method calls act like transitions of a state-machine.

Then in Sections 3.2 and 3.3, the common Java code for the classes is annotated with the appropriate specification for each tool capturing the intended object protocols.

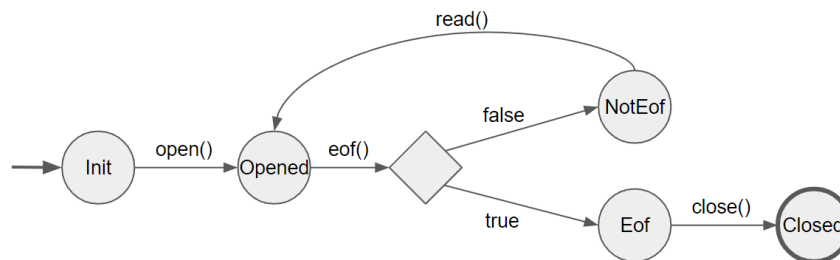
Finally, in Section 3.4, we present annotated client code which uses the object code defined before, and answer the **RQ**: can the tools check protocol compliance and completion, even when objects are shared in a collection?

All code implemented is available online. Throughout the text, hyperlinks in blue point directly to the lines of code relevant to what is being discussed in order to help the reader and to avoid the need to download code. Nonetheless, crucial code parts are presented in listings and discussed in detail. Additionally, a thorough discussion of the implementation is also online for the interested reader.

³ <https://www.key-project.org/docs/UsingKeyBook/>

3.1 Running examples

In this study, the main objects with protocol we consider are file readers. Their usage protocol is shown in Figure 2. Circles represent states, arrows denote transitions performed by method calls, and diamonds represent a decision based on the return value of the preceding call. The initial state is marked with an incoming arrow without an outgoing state. The final state is marked with a thicker border. We refer the reader to [60, 61] for more information about this kind of automata, called *Deterministic Object Automata*, and a tool generating those.



■ **Figure 2** File reader's protocol.

According to the file reader's protocol, the `open` method must be initially called. Then, one must call the `eof` method to check if the end of the file was reached. While it returns `false`, `read` calls are allowed. Otherwise, the whole file was read and the `close` method needs to be called to terminate the protocol. A usage example of a file reader exhibiting protocol compliance and completion is shown in Listing 7.

■ **Listing 7** FileReader's usage example.

```

1 FileReader f = new FileReader("file.txt");
2 f.open();
3 while (!f.eof()) {
4   f.read();
5 }
6 f.close();

```

To track the number of bytes still available to be read, the reader has an internal `remaining` field. Additionally, we may use a `state` field to track the current state of the object. This may be seen as unnatural and superfluous, but when there is no primitive notion of `typestates` in the language, it is unavoidable. One example of this encoding of protocols being employed is in the `Casino` contract presented in the `VerifyThis` event.⁴ Exemplifying code is presented in Listing 8. Assume that methods prefixed with `file_` perform actions on the file system. Please note that in the tool specific implementations, files are not actually read, so as to simplify the code for demonstration purposes.

■ **Listing 8** FileReader's code.

```

1 class FileReader {
2   String filename; State state; int remaining;
3
4   FileReader(String name) {
5     filename = name;
6     state = INIT;
7   }

```

⁴ <https://verifythis.github.io/casino/>

40:10 On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage

```
8
9 void open() {
10     assert (state == INIT);
11     remaining = file_size(filename);
12     state = OPENED;
13 }
14
15 boolean eof() {
16     assert (state == OPENED);
17     return remaining == 0;
18 }
19
20 byte read() {
21     assert (state == OPENED && remaining > 0);
22     remaining--;
23     return file_byte_read(filename);
24 }
25
26 void close() {
27     assert (state == OPENED && remaining == 0);
28     file_close(filename);
29     state = CLOSED;
30 }
31 }
```

After implementing and specifying the file readers, we implement a collection to store these. The challenge is then to verify the collection and statically track the different states of the stored file readers, while ensuring protocol compliance and completion.

For our collection, we implement a singly-linked-list, meaning that each node has a reference only to the next node. This list has two fields: `head` and `tail`. The former points to the first node, the latter points to the last node, as it is commonly implemented in imperative languages. Items are added to the `tail` and removed from the `head`, following a FIFO discipline. Having a `tail` field is crucial for efficiency, avoiding the need to iterate all the nodes before adding a new node to the end. Code for it is presented in Listing 9.

■ Listing 9 Linked-list's code.

```
1 class Node<T> {
2     T value; Node next = null; Node(T v) { value = v; }
3 }
4
5 class LinkedList<T> {
6     Node<T> head = null; Node<T> tail = null;
7
8     void add(T value) {
9         if (head == null) {
10             head = tail = new Node(value);
11         } else {
12             tail.next = new Node(value);
13             tail = tail.next;
14         }
15     }
16
17     T remove() {
18         assert (head != null);
19         T value = head.value;
20         if (head == tail) {
21             head = tail = null;
22         } else {
23             head = head.next;
24         }
25         return value;
26     }
27 }
```

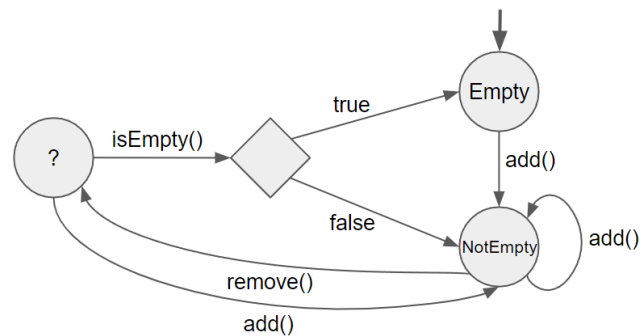
```

27
28   boolean isEmpty() { return head == null; }
29
30   LinkedListIterator<T> iterator() {
31       return new LinkedListIterator(head);
32   }
33 }

```

We believe implementing a linked-list is particularly relevant for a number of reasons. Besides being a very common data structure, and quite simple in nature, its use of pointers often creates challenges for type systems without support for *deductive reasoning*. For example, in a type system with a strict ownership discipline, it is difficult to deal with the aliasing between the `tail` field and the second to last node's `next` field.⁵ Additionally, matching the concrete structure of the collection (i.e. the linked nodes) with the abstract representation, to be able to track the states of the stored objects, usually requires ghost code and (again) deductive reasoning, as we will observe.

Although the number of values stored in the linked-list is arbitrary, we can define a finite protocol (Figure 3) that over-approximates the possible states: the list may be empty, which means that we are only allowed to add new values; or the list may be not empty, which means that we can also remove at least one value.⁶ With just these two states it is unknown if the list becomes empty or not after removing a value, so we need to encode this uncertainty with an additional state and use the `isEmpty` method to check the emptiness of the list.



■ **Figure 3** List's protocol.

Similarly, we can define a protocol for an iterator over the linked-list (Figure 4) with three states: one indicating that there are values to retrieve with the `next` method, another to specify that we reached the end of the list, and finally, a state to encode the uncertainty between the previous two, where the `hasNext` method may be used to check if there are still values to return. Code for the iterator is presented in Listing 10.

■ **Listing 10** Iterator's code.

```

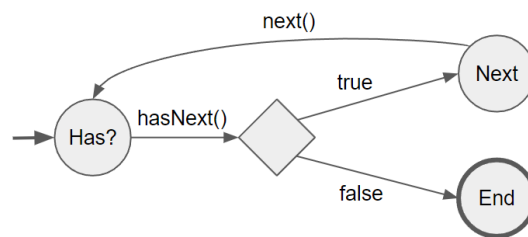
1   class LinkedListIterator<T> {
2       Node<T> curr;
3       LinkedListIterator(Node head) { curr = head; }

```

⁵ For instance, in Rust, the ownership discipline prevents one from creating linked-lists, unless `unsafe` code is used. `GhostCell`, a recent solution to deal with this, allows for internal sharing but the collection itself still has to respect the discipline [62]. It uses `unsafe` code for its implementation but was proven safe with separation logic.

⁶ Context-free session types can be used to describe protocols which are not limited by the expressiveness of regular languages [2].

40:12 On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage



■ **Figure 4** Iterator's protocol.

```
4
5  boolean hasNext() {
6      return curr != null;
7  }
8
9  T next() {
10     assert (curr != null);
11     T value = curr.value;
12     curr = curr.next;
13     return value;
14 }
15 }
```

An example combining the three classes is shown in Listing 11. In this example, a number of file readers in their initial state is added to the list (line 6). This collection is then passed to the auxiliary method `useFiles` (line 7) which iterates through all the readers and, for each one, follows their protocol to the end. The properties of protocol compliance and completion hold for this program example.

■ **Listing 11** Usage example in code.

```
1  void main() {
2      LinkedList list = new LinkedList();
3      FileReader f1 = new FileReader("a");
4      FileReader f2 = new FileReader("b");
5      FileReader f3 = new FileReader("c");
6      list.add(f1); list.add(f2); list.add(f3);
7      useFiles(list);
8  }
9
10 void useFiles(LinkedList<FileReader> list) {
11     LinkedListIterator it = list.iterator();
12     while (it.hasNext()) {
13         FileReader f = it.next();
14         f.open();
15         while (!f.eof()) f.read();
16         f.close();
17     }
18 }
```

We will now report on our experience with each tool detailing the specification and proof effort required, keeping the issues of protocol compliance and completion in mind. In summary, we successfully verify the file reader class (Listing 8) and its use (Listing 7) in all four tools. The linked-list and iterator implementations (Listing 9 and 10), as well as their usages (Listing 11), are accepted by all except Plural. Protocol completion can be guaranteed with workarounds in all tools except Plural.

3.2 File reader specification

VeriFast

To implement the file reader, we use pre- and post-conditions in all public methods indicating the required and the ensured states after the call (example in List. 12). The fields of this object are `state` (to keep track of the current state), and `remaining` (to keep track of the number of bytes still to read). Every time the state needs to change, we assign to the `state` field. We use constants to identify different states, thus avoiding the use of literal numbers in specifications. To request access to the fields of the object, and to enforce that `remaining` is equal to or greater than zero, we define the `filereader` predicate.

■ **Listing 12** `close` method in VeriFast.

```

1 public void close()
2     //@ requires filereader(this, STATE_OPENED, 0);
3     //@ ensures filereader(this, STATE_CLOSED, 0);
4 {
5     this.state = STATE_CLOSED;
6 }

```

VerCors

The implementation in VerCors is very similar to the one in VeriFast, except for these differences: we do not define a predicate to abstract the contents of the object (instead we keep access to the fields exposed for practical reasons); and since VerCors supports ghost fields, we use one to track the state, using numbers to represent each state.

Plural

Given the support for tpestates, we directly define three states, *init*, *opened*, and *closed*, which refine (i.e. define a substate of) the root state `alive`, a state in which all objects are in. Additionally, we define two states, *eof* and *notEof*, refining *opened*, indicating if we have reached or not the end of the file, respectively. The file reader has a boolean field, `remaining`, indicating if there is something to read. To enforce the relation between the states *eof* and *notEof*, and the `remaining` field, we define invariants for these states. Due to a limitation, we had to simplify the `read` method, making it read the file all at once.⁷ To enforce the protocol, we declare for each method the required and ensured states as well as the permissions needed to perform each call. The `open`, `read`, and `close` methods require `unique` permission to the receiver object. *Full* permissions would be enough except for the possibility of concurrent accesses, which would require methods to be `synchronized`. The `eof` method only needs an *immutable* permission guaranteeing that we are in the *opened* state, and returns a boolean value indicating if the end of the file was reached.

KeY

We also model the protocol with pre- and post-conditions in methods. The state is tracked using an integer ghost field. As in VeriFast and VerCors, `remaining` stores the number of bytes left to read, and we enforce the value to be equal or greater than zero with an

⁷ Ideally, `remaining` would store an integer, but the syntax `remaining == 0` does not seem to be supported in the invariants. In consequence, we cannot model the arbitrary number of bytes to read.

40:14 On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage

invariant. The file reader has a footprint, composed by the `state` and `remaining` fields, and each of its methods specifies which fields may be modified, according to the dynamic frames technique [40]. To verify this class, we have to prove each method correct, according to each specification, and the fact that nothing changes the footprint. Given its simplicity, all proofs were automatically done using KeY's default strategy.

Evaluation

As expected, we successfully modelled a protocol for a file reader in all four tools: in Plural, the implementation was mostly straightforward given the support for tpestates, but annotations were required in all methods; in VeriFast, VerCors, and KeY, we used method contracts, which also required some annotation burden. Thus, we motivate the need for more natural ways to specify protocols, for example, via automata, which helps the programmer visualize and design the protocol.

3.3 Linked-list and iterator specifications

VeriFast

The linked-list implementation is adapted and extended from a C implementation available online. One key difference from the aforementioned C code is that when the linked-list is empty, the `head` and `tail` fields have `null` values, instead of pointing to a dummy node. This matches common implementations and makes verification more challenging because we have to avoid null pointer errors.

To model the structure of the list, we define a predicate that holds access to the `head` and `tail` fields and of all the nodes in the list (List. 13). The only input parameter is the reference to the linked-list. The output parameters are the references to the head and tail, and a ghost list to reason about the values in the list in an abstract way (line 1). This ghost collection will be crucial to track the different states of the file readers stored in the list. Lines 3 and 4 ensure that if one of the fields is `null`, the other is also `null`, and the list is empty. To ease the addition of new elements to the list, we request access to the sequence of nodes between the head (inclusive) and the tail (exclusive), through the `lseg` predicate, and then keep access to the tail node separately (line 5). The `node` predicate holds the permissions to the `next` and `value` fields of a given node. Note that we do not hold permission to the fields of the values stored. This is to allow them to change independently of the linked-list.

■ **Listing 13** `l1ist` predicate in VeriFast.

```
1 predicate l1ist(LinkedList obj; Node h, Node t, list<FileReader> list) =
2   obj.head |-> h &&& obj.tail |-> t &&&
3     h == null ? t == null &&& list == nil :
4     t == null ? h == null &&& list == nil :
5     lseg(h, t, ?l) &&& node(t, null, ?value) &&&
6     list == append(l, cons(value, nil)) &&& list != nil;
```

The implementation of the `remove` and `isEmpty` methods is straightforward requiring only the unfolding and folding of the `l1ist` and `lseg` predicates a few times. The `add` method requires an auxiliary lemma (List. 14) stating that if we have a sequence of nodes plus the final node, and we append another node in the end, we get a new sequence with all the nodes from the previous sequence, the previous final node, and then the newly appended node.

■ **Listing 14** `add_lemma` lemma in VeriFast.

```
1 lemma void add_lemma(Node n1, Node n2, Node n3)
2   requires lseg(n1,n2,?l) &&& node(n2,n3,?value) &&& node(n3,?n4,?v);
3   ensures lseg(n1,n3,append(l,cons(value,nil))) &&& node(n3,n4,v);
```

To implement the iterator, we define a `iterator` predicate which holds access to the current node field and all the nodes in the linked-list. Then, we split the permissions to the nodes in two parts (List. 15): half of the permissions preserves the structure of the list (line 4), and the other half holds the view of the iterator (line 5): a sequence of nodes from the head (inclusive) to the current node (exclusive); and a sequence from the current node (inclusive) to the final one. Both parts allow us to reason on the values already seen, and the values still to be seen. This split occurs when the iterator is created. After iterating all the nodes, the full permission to the nodes needs to be restored to the list, which is done via an auxiliary lemma.

■ **Listing 15** `iterator_base` predicate in VeriFast.

```

1 predicate iterator_base(LinkedList javalist, Node n;
2   list<FileReader> list, list<FileReader> a, list<FileReader> b) =
3   [1/2] javalist.head |-> ?h &&& [1/2] javalist.tail |-> ?t &&&
4   [1/2] llist(javalist, h, t, list) &&&
5   [1/2] lseg(h, n, a) &&& [1/2] nodes(n, b) &&& list == append(a, b);

```

The implementation of the `hasNext` method is straightforward. The implementation of the `next` method requires unfolding and folding predicates, the use of a lemma showing that the `append` function is associative (result already available in VeriFast), and the `iterator_advance` lemma, which helps us advance the state of the iterator, moving the just retrieved value from the “to see” list to the “seen” list (List. 16).

■ **Listing 16** `iterator_advance` lemma in VeriFast.

```

1 lemma void iterator_advance(Node h, Node n, Node t)
2   requires [1/2] lseg(h, n, ?a) &&& [1/2] node(n, ?next, ?val1) &&&
3     [1/2] nodes(next, ?b) &&& [1/2] lseg(h, t, ?list) &&&
4     [1/2] node(t, null, ?val2);
5   ensures [1/2] lseg(h, next, append(a, cons(val1, nil))) &&&
6     [1/2] nodes(next, b) &&& [1/2] lseg(h, t, list) &&&
7     [1/2] node(t, null, val2);

```

VerCors

The implementations of the linked-list and iterator closely follow the VeriFast’s ones, with just some differences. Given that predicates in VerCors do not support output parameters, we have predicates to request access to the needed memory locations, and then methods to build the ghost lists that allow us to track the values. Additionally, we use `given` and `yields` clauses in the methods to receive and return the necessary lists (lines 1-2 of List. 17). Instead of using these clauses in methods, we would have preferred to rely on ghost fields storing those lists. Unfortunately, it seems one cannot reason about the old value of a field if its permission is inside a predicate. Using inline unfolding does not seem to work.

■ **Listing 17** `remove` method’s contract in VerCors.

```

1 given seq<FileReader> oldList;
2 yields seq<FileReader> newList;
3 requires state(oldList) ** |oldList| > 0;
4 ensures state(newList) ** newList == tail(oldList);

```

Linked-lists and iterators have been implemented before for VerCors [14] but, to our knowledge, there is no list implementation that uses a `tail` field, preferring instead a recursive approach, with the “head” being the first value, and the “tail” being the rest of the list.

Plural

For the linked-list, we adapt a stack example from Plural's repository.⁸ Naturally, we make the appropriate changes since our linked-list follows a FIFO discipline, while a stack follows a LIFO one. Since objects in Plural should be associated with tpestates, both our `Node` and `LinkedList` classes have protocols.

In the `Node` class, we define two orthogonal state dimensions, *dimValue* and *dimNext*, which handle the `value` and `next` fields, respectively. In *dimValue* there are two states, *withValue* and *withoutValue*, which indicate if the node has permission to the stored value or not. In *dimNext* we have states *withNext* and *withoutNext*, which say if the node has permission to the next node or if `next` is `null`. State dimensions avoid the need to reason about all the combinations of having (or not) a value and having (or not) a next node. Since direct field accesses are disallowed, we define getter and setter methods for both fields.

In the `LinkedList` class, we define two states: the empty and the non-empty. When it is empty, the `head` and `tail` fields are `null`. When it is not empty, `head` and `tail` are not `null` and there is unique permission to the first node, which is pointed by `head`. Since the `head` points to the next node, and so on, we should have the required chain of nodes that builds the linked-list. Adding and removing values from the list require *unique* permission to it.

Unfortunately, Plural did not accept either implementation. With regards to the `Node` class, we had errors in all the methods indicating that the receiver could not be packed (i.e. coerce from the concrete field view of the class to the abstract tpestate view [24]) to match the state specified by the `ensures` annotation parameter. Additionally, the invariant for the *withValue* state had an error stating that the parametric permission kind we specified was unknown, even though that was introduced with the appropriate annotation. In fact, we did the same for the `LinkedList` class and we did not get these kinds of errors.

With regards to `LinkedList`, the only errors reported were in the `add` method. To understand why, consider the case in which the list is not empty. In this case, the `tail` is non-null, and we must call `setNext` on it to append a new node (line 8 of List. 18). However, we do not have permission to do that. For this to work, we would need to have permission to the last node that is owned by the second to last node. Unfortunately, we could not perform such transferring of permissions. An alternative solution could be to use *share* permissions instead of *unique* ones in the nodes. But this would require locking when accessing them, because of the possibility of thread concurrency. Furthermore, we would lose track of the memory footprint used by the list (since *share* permissions allow for unrestricted aliasing). This can be an issue if we want to track all the references and ensure statically that all resources are freed at end. Given this, we did not attempt to implement the iterator.

■ Listing 18 `LinkedList`'s `add` method in Plural.

```

1 @Unique(requires="alive", ensures="notEmpty", use=Use.FIELDS)
2 public void add(@PolyVar(value="p", returned=false) T value) {
3     @Apply("p") Node<T> n = new Node<T>(value);
4     if (head == null) {
5         head = n;
6         tail = n;
7     } else {
8         tail.setNext(n);
9         tail = n;
10    }
11 }

```

⁸ File `pluralism/trunk/PluralTestsAndExamples/src/edu/cmu/cs/plural/polymorphism/ecoop/Stack.java`

KeY

The linked-list implementation is heavily inspired in a tutorial by Hiep et al. which implements a doubly-linked-list [29]. We declare several fields: `head` and `tail`; `size`, to count the number of values; `nodeList`, containing a sequence of nodes; and `values`, containing a sequence of values. The `nodeList` and `values` fields are ghost fields. As for the file reader, we define the linked-list's footprint, composed by its fields and the fields of all the nodes (line 5 of List. 19). We also specify that the footprint itself only changes if the nodes sequence changes (line 2), and that the list's invariant only depends on the locations in the footprint (line 3). The proof of the former was generated automatically by KeY using the default strategy. The proof of the latter required some interactivity to guide the proof. Note that the footprints of the values are not part of list's footprint.

■ Listing 19 List's footprint in KeY.

```

1 public model \locset footprint;
2 accessible footprint: nodeList;
3 accessible \inv: footprint;
4 represents footprint = size, head, tail, nodeList, values,
5   (\infinite_union \bigint i; 0 <= i < nodeList.length; ((Node)nodeList[i]
   ).*);

```

The list's invariant is the most verbose part of the specification. First, we specify that `size` is equal to the number of nodes, which is then equal to the number of values. Then we enforce that the values in the list are not null. We also use an existential quantifier, indicating that in each position of the sequences, elements exist (lines 1-2 of List. 20). This is necessary because KeY treats sequences in a way where they may occasionally contain not-yet-created objects. Additionally, since the sequence declarations do not enforce the type of their elements, we need to do it explicitly, either using an `instanceof` operator, or using an existential quantifier. Furthermore, we need to cast the result of accessing a position in a given sequence. Following that, we have to take into account that the list may be empty. So, we define that either the nodes sequence is empty, and the `head` and `tail` fields are null, or the nodes sequence is not empty, and the `head` points to the first node, and `tail` points to the last one (lines 3-6). To ensure we have a linked-list, we enforce that the `next` field of each node points to the following node in the sequence (lines 7-8). We also enforce that all the nodes are distinct (lines 9-12). Finally, we specify that each value in the values sequence corresponds to the value stored in each node in the same position.

■ Listing 20 List's invariant in KeY.

```

1 (\forallall \bigint i; 0 <= i < values.length;
2   (\exists FileReader f; f == values[i] && f != null)) &&
3 ((nodeList == \seq_empty && head == null && tail == null)
4   || (nodeList != \seq_empty && head != null && tail != null &&
5     tail.next == null && head == (Node)nodeList[0] &&
6     tail == (Node)nodeList[nodeList.length-1])) &&
7 (\forallall \bigint i; 0 <= i < nodeList.length-1;
8   ((Node)nodeList[i]).next == (Node)nodeList[i+1]) &&
9 (\forallall \bigint i; 0 <= i < nodeList.length;
10  (\forallall \bigint j; 0 <= j < nodeList.length;
11   (Node)nodeList[i] == (Node)nodeList[j] ==> i == j
12  )) && ...

```

The iterator has several fields: `list`, a reference to the list; `curr`, the current node; `index`, the position of the current node in the nodes sequence; `seen`, the sequence of values already iterated; and `to_see`, the sequence of values still to be iterated. The `index`, `seen`, and `to_see` fields are ghost fields. As usual, we define the iterator's footprint, composed only

by its fields (line 4 of List. 21). We also specify that the footprint itself does not change (line 2), and that the iterator’s invariant depends on its footprint and on the list’s footprint (line 3). The proof of the former was done automatically. For the proof of the latter, KeY’s default strategy was not enough. The reason for this was that KeY was applying multiple “cut” tactics to try to close the proof for each possible value of `size`. Nonetheless, it was mostly straightforward to guide the proof. We just had to use the “observerDependency” tactic to establish that the iterator’s invariant does not change in the presence of heap updates on locations that do not belong to its footprint or the list’s footprint.

■ **Listing 21** Iterator’s footprint in KeY.

```

1 public model \locset footprint;
2 accessible footprint: \nothing;
3 accessible \inv: footprint, list.footprint;
4 represents footprint = list, curr, index, seen, to_see;

```

In the iterator’s invariant we first specify that `index` is a value between zero and the number of values. This number may be equal to the number of values if and only if we have already iterated through all values. Then we enforce that the values in both sequences are not `null`. Following that, we define that the `seen` sequence corresponds to the values already seen, from position zero (inclusive) to `index` (exclusive), and that `to_see` corresponds to the values to see, from position `index` (inclusive) to the end. Since `curr` points to the current node, we map it to position `index` in the nodes sequence, or we specify that it is `null`, when iteration is done. Finally, we assert that the list’s invariant holds.

Regarding the linked-list, verifying the constructor, `add`, and `iterator` methods required only the default strategy, but for the `remove` method, some interactivity was needed, namely to show that the first value was the value of the `head`. Regarding the iterator, we had to guide the proof of the constructor, mostly to establish the invariants of the iterator and list, since KeY was applying “cut” multiple times, as before. The `hasNext` method was verified automatically with the default strategy. To verify the `next` method, we had to prove that: 1. only the list’s and iterator’s footprints are accessible; 2. the post-condition holds after execution; 3. and that only the iterator’s footprint is modified. These proof requirements required a lot of work, likely because of the relation between `index` and `curr`, which was probably not obvious to the default strategy. Examples of goals which required some effort to prove were: showing that the value of the current node was the first value in the “to see” sequence, proving that such a value was a file reader, and that the new sequences respected the invariant (after the current value was moved to the “seen” sequence).

Evaluation

In VeriFast, as well as in VerCors, the expressiveness of the logic allowed us to specify a linked-list and an iterator. However, deductive reasoning was often required. In our experience, we spent more time in proving results than in writing code, having to unfold and fold predicates very often, revealing their definitions, having to define multiple lemmas, and insert assertions to guide proofs. In VeriFast, we had to write about 160 lines of lemmas. In VerCors, we wrote about 100 lines. Some of the time spent in VerCors with the proofs was reduced because we could reuse the experience we had with VeriFast.

In Plural, we were not able to implement a linked-list. We believe the support for logical predicates would be necessary to be able to specify structures with recursive properties. Furthermore, as far as we can tell, there is no support for parametric tpestates, even though there is for fractional permissions, which could potentially allow one to model a list with objects in different states that evolve.

In KeY, we were able to implement and verify a linked-list and an iterator. Although KeY supports interactivity, together with useful macros and a high degree of automation, we spent some time proving properties about the heap. For example, we often had to prove that the footprints of two objects were disjoint, which means we also had to account for possible changes in the footprints themselves, which became very cumbersome. To do this, we had to make the footprints public so that they could be opened in proofs, otherwise we are not sure if we would have been able to finish the proofs. We believe this motivates the need to be able to prove heap and functional properties separately.

Given the differences between the approaches presented, we believe simply comparing lines of specification would not provide a meaningful comparison. Thus, we provide a qualitative evaluation. In summary, we observe that these methods require an important effort especially when one is learning the approaches. Not surprisingly, in VeriFast, VerCors, and KeY, the specification took more space than the code, and was usually verbose. VeriFast and VerCors also required a substantial amount of annotations to guide the proofs. With KeY, the space, usually needed for proofs in the other two tools, was replaced by the time spent proving results interactively. Even if it turns out that with training, it is not that hard to specify, implement, and verify the examples, it is certainly time consuming. We believe this motivates further study on what we can delegate to static analysis to ease this effort. Plural has less expressive power than the other tools, so it makes sense that the annotation effort was low.

3.4 RQ evaluation

We are able to produce examples of file readers usage, as presented in Section 3.1, where we successfully ensure that the protocol is followed (i.e. only the allowed methods in each state can be called), in all four tools: in VeriFast, VerCors, and KeY, thanks to the pre- and post-conditions; and in Plural, thanks to the tpestate abstraction directly supported. We can also ensure protocol compliance when different file readers are stored within a linked-list, in VeriFast (List. 22), VerCors, and KeY, but with significant annotation and proof effort, as observed in the examples produced.

■ **Listing 22** useFiles specification in VeriFast.

```

1 requires list != null &&& llist(list, _, _, ?1) &&& tracker(length(1))
   &&& foreachp(1, INV(FileReader.STATE_INIT));
2 ensures list != null &&& llist(list, _, _, 1) &&& tracker(0) &&& foreachp
   (1, INV(FileReader.STATE_CLOSED));

```

In the rest of this section, we focus our presentation on the crucial property of **protocol completion**, which guarantees that if and when the program terminates, all the objects are in the final states of their protocols.

VeriFast

To ensure that all file readers created through the lifetime of the program reach the end of their protocol, we define a **tracker** predicate in a *.javaspec* file which keeps hold of the number of open file readers (List. 23). This proposed solution is based on a private exchange with Jacobs [37]. Then, we augment the file reader's specification to increment this counter in the constructor, and decrement the counter in the **close** method. Finally, since we want to guarantee protocol completion for all created objects, we assert in the pre-condition and post-condition of the **main** method that the counter should be zero, given that **main** is the starting (and ending) point of Java programs.

■ **Listing 23** tracker.jvaspec file.

```

1 predicate tracker(int count);
2
3 lemma void increment_tracker();
4   requires tracker(?n);
5   ensures tracker(n + 1);
6
7 lemma void decrement_tracker();
8   requires tracker(?n);
9   ensures tracker(n - 1);

```

Unfortunately, it is possible to fail to ensure protocol completion if the programmer is not careful. Firstly, one could forget to increment and decrement the counter when the `typedstate-object` is initialized and when its protocol finishes, respectively. Secondly, if one forgets to include the post-condition in the `main` method, protocol completion will not be actually enforced. So, we can guarantee protocol completion but only if the programmer does not fall for these “traps”. Here we see that ghost code is useful to check properties, but if such code is not correctly connected with the “real” code, then the property we desired to establish is not actually guaranteed.

VerCors

To ensure protocol completion, we follow the previous idea, but instead of a “global counter” defined through a predicate written in a specification file, we create a `FileTracker` object which keeps hold of the number of open file readers using ghost code. When we augment the file reader’s implementation to increment and decrement the counter in the appropriate methods, we also have to pass the tracker using the `given` directive. Again, it is possible to fall for the same “traps”: forgetting to increment and decrement the counter, and forgetting to add the post-condition to the `main` method.

Since VerCors supports quantifiers, one could think of quantifying over all file readers and ensuring they are all closed. Unfortunately, we would be quantifying over all possible file readers, not just the ones actually allocated on the heap.

Plural

Although the `typedstate` abstraction is directly supported, protocol completion is not guaranteed since permissions may be “dropped”, as seen in List. 24, where a unique permission for an object is received but not used, without any error being reported. This was not an issue in other tools, even though leaking resources is permitted, because the support for deductive reasoning allowed us to count the number of active objects.

■ **Listing 24** Dropping file reader in Plural.

```

1 void droppingObject(
2   @Unique(requires="opened", returned=false) FileReader f) {}

```

As far as we can tell, Plural’s specification language is based on linear logic, which would imply that this would not be permitted. However, we understand why this is the case in Java, since it is common for one to stop using an object and letting the garbage collector reclaim memory. Nonetheless, we believe that ensuring protocol completion is crucial for `typedstate-objects`, to ensure that important method calls are not omitted and resources are freed (e.g. closing a socket).

KeY

To ensure that all file readers reach the end of their protocol, we define a contract for the `main` method such that for all file readers created at some point in the program, they are in the final state (line 2 of List. 25). Since KeY is not aware of the objects created or not before, we define as pre-condition that no file readers exist when `main` is called (line 1). This works because quantifiers in KeY only reason over objects in the heap. Given that `main` is the first method called in a Java program, this requirement is actually an assumption.

■ **Listing 25** `main` method’s contract with protocol completion in KeY.

```
1 requires !(\exists FileReader f; true);
2 ensures (\forallall FileReader f; f.state == FileReader.STATE_CLOSED);
```

Thanks to first-order dynamic logic, we can use quantifiers to specify that all existing file readers should have their protocol completed. Although this seems powerful, we have to adapt all other methods to specify that no new file readers are created inside. This is necessary because it would be possible for methods to create file readers only available in the scope of their execution, which would exist in the heap as created objects, but for which we would know nothing about. This post-condition, written as `!(\exists FileReader f; \fresh(f))`, was added in all needed methods. In the file reader’s constructor, we also had to say that `f` was different from `this`, since the newly created reference is fresh.

Evaluation

In the context of tpestates, checking for protocol completion is crucial to ensure that necessary method calls are not forgotten and that resources are freed, thus avoiding memory leaks. Unfortunately, that concept is not built-in in any of the logics employed by all four tools. We believe that protocol completion should be provided directly by the type system and the programmer should not be required to remember to add this property to the specification.

One workaround we found for VeriFast and VerCors was to have a counter that keeps track of all tpestated-objects which are not in the final state. This requires keeping hold of the aforementioned tracker in specifications, which can be a huge burden in bigger programs. Ensuring protocol completion could be embedded in separation logic and such a feature could even be possible in VeriFast. Given that VeriFast supports leak checking, one would just need to incorporate notions of tpestates and ensure that leaking would only be allowed when objects are in their final states. In C, one would also need to enforce that claimed memory is freed. In Java, leak checking would need to be enabled for tpestated-objects.

For Plural, we did not find a way to ensure protocol completion. This could be supported by asking the programmer to indicate which state of a given object is the final one and only allowing permissions for *ended* objects to be “dropped”.

In KeY, we made use of the support for universal quantifiers and reasoning on heap-dependent expressions. However, verifying the code against that specification can be cumbersome, and requires augmenting the specifications of all other methods, ensuring no untracked objects are added to the heap.

4 General assessment of the tools

In this section, we summarize our views about the tools, using the knowledge gained from our experiments, and provide suggestions of what could be improved.

VeriFast

Separation logic, fractional permissions, and predicates, allow for rich and expressive specifications that make it possible to verify complex programs. However, deductive reasoning is often required when the specifications are more elaborate, as we have seen when implementing the linked-list (Section 3.3). This is tedious and can be a barrier to less experienced users. Although VeriFast’s IDE provides a way for one to look at each step of a proof when an error is discovered, we believe that, at least in part, having a way to guide proofs (like one can do with proof assistants such as Coq), would improve the user experience even more by: (1) avoiding having to insert proof guiding assertions in the code implementation itself (allowing for more separation of concerns); and (2) avoiding the need to rerun the tool every time that occurs. In other words, when guiding the proofs, one would get immediate feedback. Nonetheless, the IDE experience was still very useful in helping us prove several results.

Checking for protocol completion is crucial to ensure that essential method calls are not omitted and that resources are freed. Unfortunately, that concept is not built-in in separation logic (Section 3.4). Given that VeriFast supports leak checking, one would just need to ensure that leaking would only be allowed when objects are in their final states.

VerCors

As in VeriFast, rich and expressive specifications are supported, but deductive reasoning is (again) required. As we noted before, this can be tedious, as highlighted by the time and lines of code we needed to prove results. Unfortunately, VerCors has no interactive experience so we had to practice “trial and error” more often, guessing what could be wrong and rerunning the tool every time we changed the code.

In terms of user experience, we think allowing for more separation between lemma and predicate functions from code, instead of forcing these to belong to classes as static methods, would help improve readability, as others have also noted [31]. We also believe the tool could be more efficient: since specifications are self-framing (i.e. only depend on memory locations that they themselves require to be accessible) and the checking process is modular, VerCors could cache some results to avoid re-checking parts of the code that were not modified. We also noticed that if we unfolded a predicate on which some truth depends on, that knowledge would be lost. For example, the knowledge of the values stored in a sequence of nodes depends on the permissions to those nodes, available in the `nodes_until` predicate. In principle, unfolding this predicate should not invalidate the available information, but it does. To workaround this, we had to use fractional permissions to keep hold of some fraction of the original predicate, and only unfold the other fractional part.

Comparison between VeriFast and VerCors

Given the similarities between VeriFast and VerCors, we believe it is very relevant to provide a comparison between both.

With respect to specifying access to memory locations, VeriFast only supports the points-to assertions of separation logic, while VerCors also supports permission annotations, inspired by Chalice [44, 45], allowing us to refer to values in variables without the need to use new names for them, which was very useful when writing the specifications. Furthermore, VerCors has built-in support for quantifiers, many different abstract data structures, and ghost code, which VeriFast does not. We used a fair amount of ghost code in VerCors. Nonetheless, VeriFast supports the definition of new inductive data types, fixpoint functions, higher-order predicates, and counting permissions, which VerCors does not. Unfortunately, VerCors does

not support generics in Java. Regarding VeriFast, as Bart Jacobs points out, at the time of writing, “support for Java generics in VeriFast is in its infancy”.

Both provide support for fractional permissions, which we have used. However, this model only allows for read-only access when data is shared. In consequence, either locks are required to mutate shared data (even in single-threaded code, where they are not really necessary, resulting in inefficient code), or a complex specification workaround is needed. We believe that the specifications and code should focus on the application’s logic, and the need to modify them to help the verifier should be avoided as much as possible. VerCors lacks support for counting permissions, which would allow permissions to be split in other ways.

Finally, we missed the support for output parameters which VeriFast has. To reproduce the same concept in VerCors, we had to add ghost parameters in many methods and explicitly pass values for those parameters when calling such methods. For example, when working with the linked-list, we kept track of the sequence of values in the list through ghost code, and always had to pass that sequence to each called method.

Plural

The rich set of access permissions allows objects’ state to be tracked even in the presence of aliasing, and permits read/write and write/write operations, thanks to state guarantees. Nonetheless, we could not specify structures such as the linked-list with double handle (i.e. with head and tail fields), likely because of the lack of support for logical predicates. Furthermore, to our knowledge, there is no support for parametric typestates.

The use of *share* permissions allows for unrestricted aliasing. Nonetheless, state assumptions need to be discarded because of the possibility that there might be other threads attempting to modify the same reference. Although this thread-sharedness approximation is sound, it forces the use of synchronization primitives even if a reference is only available in one thread. Beckman et al. discuss the possibility of distinguishing permissions for references that are only aliased locally from references that are shared between multiple threads, allowing access to thread-local ones without the need for synchronization [7]. But as far as we know, the idea was not realized.

Furthermore, there is no built-in guarantee of protocol completion. This could be provided by only permitting permissions for *ended* objects to be “dropped” (Section 3.4).

KeY

The use of JML for specifications, a language for formally specifying behavior of Java code, used by various tools, reduces the learning curve for those that already know JML. Furthermore, KeY supports a great number of Java features, allowing one to verify real programs considering the actual Java runtime semantics. Nonetheless, generics are not supported, although there is an automated tool to remove generics from Java programs, which can then be verified with KeY. Because of its focus on Java, KeY is not overly suitable for the verification of algorithms that require abstracting away from the code [20].

One important aspect that makes KeY stand out from other tools is the support for interactivity, which allows the programmer to guide the proofs. This is an aspect that we missed when experimenting with other tools. Additionally, KeY provides useful macros and a high degree of automation, as well as support for SMT solvers, such as Z3. We used Z3 often to more quickly close provable goals, specially those involving universal quantifiers.

Since KeY’s core is based on first-order dynamic logic, one can express heap-dependent expressions: the heap is an explicitly object in the logic. Although this allows for much expressiveness, it often becomes very difficult to verify programs, as our experience has shown

(Section 3.3). We think that a variant of KeY that would instead use separation logic, to abstract away the heaps and the notion of disjointness, would be very helpful, improving readability and reducing verbosity.

Nonetheless, there are probably other alternatives to separation logic that would help in solving the aforementioned issues. In a private conversation with KeY’s group leaders, they point out that the connectives of separation logic “would get in the way of automation”, a crucial feature of KeY. For example, it seems that “the heap separation rule tends to split proofs too early” [35]. Furthermore, they mention that the problems we encountered could be summarized in two main points: (1) insufficient abstract specification primitives; (2) inability to prove heap and functional properties separately, in a modular fashion. KeY’s team is aware of these issues and will address them in the future (at the time of writing).

As we pointed out above, we believe separation logic together with resource leaking prevention (except for objects with completed protocol), could be used to ensure protocol completion without the need for adding extraneous specifications. This would be another reason why we believe separation logic would be preferable over first-order dynamic logic, but it is possible there are other alternatives.

Finally, we enjoyed the user experience and appreciated that KeY comes with examples to experiment with. Nonetheless, at the moment of writing, we believe there is room for improvement. For more details, we refer the reader to a thorough discussion of the issues we found and suggestions for improvements, which we shared with KeY’s team.

5 Related work

Penninckx et al. develop an approach to verify input/output properties of programs [54]. They encode I/O behavior using abstract permission-based predicates implemented in VeriFast. The technique ensures that a program only performs the allowed I/O operations. Additionally, it guarantees a terminated program has performed all desired operations with a post-condition specifying the final state a program should be found in. Later, Jacobs presented an approach to verify liveness properties [36]. Blom et al. verify the functional behavior of concurrent software using histories, which record the actions taken by a concurrent program [15]. The technique has been integrated in VerCors and experimentally added to VeriFast. Similarly, Oortwijn integrated process algebra models [8] in VerCors to reason about functional properties of shared-memory concurrent programs, including non-terminating ones [52]. More recently, work has been developed to support the deductive verification of JavaBIP models in VerCors [12]. In these models, Java classes are considered as components where their behavior is described by finite state machines, and component interactions are specified with synchronization annotations [11]. Kim et al. propose a technique to specify protocols of Java classes by incorporating tpestates into JML [41]. When translating their extension to pure JML, multiple boolean fields for each state are declared which, when `true`, indicate the object is in that given state. Multiple fields are needed to support *state refinements* [9]. Cheon and Perumandla extend JML with a new specification clause containing a regular expression-like notation to specify the sequences of method calls allowed for a given class [21].

In this study, we focus on *sequential* examples and only present a simple protocol, which does not require a complex encoding, so the aforementioned techniques would either not be applicable or would introduce unnecessary verification overhead.

With respect to comparison studies, there are several that have been conducted. However, as far as we know, no study was previously done that focused on the verification of *protocol compliance and completion*. Nonetheless, we reference some works we found relevant to

us. Lathouwers and Huisman examine the annotation effort in several tools, including VeriFast and VerCors [42]. Hollander briefly discusses the differences between VeriFast and VerCors [31]. Boerman et al. study the way in which KeY and OpenJML treat JML specifications differently and the effort in switching between both tools [16].

6 Conclusions

In this paper, we address the **RQ** (Page 2) by reviewing four verification tools for Java. In particular, we evaluated their ability to check the correct use of objects with protocol, and if they were able to guarantee protocol completion, even when these were shared in collections (Section 3.4). Additionally, we evaluated the programmer’s effort in making the code acceptable to each, and provide suggestions for improvements (Section 4).

We were able to reach a general conclusion: stateful objects usually have protocols representing their intended usage. In the tools we have studied, protocols are not first-class entities; instead they need to be encoded with method contracts (even in Plural, annotations on methods are required). In contrast, approaches based on behavioral types, where protocols can be defined via automata [4, 47, 60], treat protocols as central, and provide a global view of the intended usage of each object. By having this model, reasoning on relevant properties becomes easier than on lower level encodings.

Now we summarize key points gathered from our experiments with each tool. Both VeriFast and VerCors support rich and expressive specifications based on separation logic: this allowed us to successfully address the **RQ**. However, deductive reasoning was often required. This is very demanding and can be a barrier to less experienced users. We believe improved interactive experiences for programmers are key to make these tools more approachable. Furthermore, fractional permissions only allow for read-only access when data is shared.

Plural is different from these tools in two major ways: it does not support logical predicates and so, specifications are less expressive in that regard, which prevented us from implementing a linked-list. Nevertheless, access permissions support more kinds of sharing, but access to thread-local shared data might require an unnatural use of locks.

KeY supports interactivity, automation, and the ability to reuse proofs. Nonetheless, the fact that heaps are mentioned explicitly in assertions made it difficult to read the hypothesis and proof goals. Additionally, we often had to show that certain footprints were disjoint. So, although we successfully answered the **RQ**, again the effort was substantial. To fully automate some proofs, KeY depends on finding the right specifications and proof search settings, which is not easy. More abstract specification primitives, and the ability to separate proofs of heap and functional properties, are crucial features to improve both readability and ease of proving results.

So, we proved protocol compliance with some effort, but protocol completion, crucial to ensure that necessary method calls are not forgotten and that resources are freed, is not directly supported by any of these tools. Although there are workarounds in some, we believe such guarantee should be supplied directly. This could be done by ensuring that no permission to an object is “dropped” unless it is in the final state.

In conclusion, this study motivates the need for lightweight methods to statically guarantee protocol compliance and completion in the presence of several patterns of sharing, like objects with protocol stored in collections, including the following features: usage protocols as the central entity defining objects’ behavior, more kinds of sharing beyond fractional permissions also avoiding the need for locks in sequential code, and better techniques to reason about permissions to heap locations.

For completion, this study could be complemented with OpenJML [22], and LiquidJava, a recent tool that integrates liquid types in Java [26].

References

- 1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification – The KeY Book – From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.
- 2 Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic lambda calculus with context-free session types. *Inf. Comput.*, 289(Part A), 2022. doi:10.1016/j.ic.2022.104948.
- 3 Davide Ancona et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016. doi:10.1561/25000000031.
- 4 Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. A Java tpestate checker supporting inheritance. *Sci. Comput. Program.*, 221, 2022. doi:10.1016/j.scico.2022.102844.
- 5 Bernhard Beckert and Reiner Hähnle. Reasoning and Verification: State of the Art and Current Trends. *IEEE Intell. Syst.*, 29(1):20–29, 2014. doi:10.1109/MIS.2014.3.
- 6 Nels E. Beckman. Modular tpestate checking in concurrent Java programs. In *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 737–738. ACM, 2009. doi:10.1145/1639950.1639990.
- 7 Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and tpestate. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–244. ACM, 2008. doi:10.1145/1449764.1449783.
- 8 Jan A. Bergstra and Jan Willem Klop. Process Algebra for Synchronous Communication. *Inf. Control.*, 60(1-3):109–137, 1984. doi:10.1016/S0019-9958(84)80025-X.
- 9 Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with tpestates. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 217–226. ACM, 2005. doi:10.1145/1081706.1081741.
- 10 Kevin Bierhoff and Jonathan Aldrich. Modular tpestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–320. ACM, 2007. doi:10.1145/1297027.1297050.
- 11 Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. Exogenous coordination of concurrent software components with JavaBIP. *Softw. Pract. Exp.*, 47(11):1801–1836, 2017. doi:10.1002/spe.2495.
- 12 Simon Bliudze, Petra van Den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java. In *26th International Conference on Fundamental Approaches to Software Engineering*, 2023.
- 13 Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Proceedings of Integrated Formal Methods*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017. doi:10.1007/978-3-319-66845-1_7.
- 14 Stefan Blom and Marieke Huisman. Witnessing the elimination of magic wands. *Int. J. Softw. Tools Technol. Transf.*, 17(6):757–781, 2015. doi:10.1007/s10009-015-0372-3.
- 15 Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. History-Based Verification of Functional Behaviour of Concurrent Programs. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods – 13th International Conference, Proceedings*, volume 9276 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2015. doi:10.1007/978-3-319-22969-0_6.
- 16 Jan Boerman, Marieke Huisman, and Sebastiaan J. C. Joosten. Reasoning About JML: Differences Between KeY and OpenJML. In *Integrated Formal Methods – 14th International Conference, Proceedings*, volume 11023 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2018. doi:10.1007/978-3-319-98938-9_3.

- 17 Richard Bornat, Cristiano Calcagno, Peter O’Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *The 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, 2005. doi:10.1145/1040305.1040327.
- 18 John Boyland. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. doi:10.1007/3-540-44898-5_4.
- 19 Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias Jakobsen, Mikkel Kettunen, and António Ravara. Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language. In *Asian Symposium on Programming Languages and Systems*, volume 12470 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2020. doi:10.1007/978-3-030-64437-6_6.
- 20 Daniel Bruns, Wojciech Mostowski, and Mattias Ulbrich. Implementation-level verification of algorithms with KeY. *Int. J. Softw. Tools Technol. Transf.*, 17(6):729–744, 2015. doi:10.1007/s10009-013-0293-y.
- 21 Yoonsik Cheon and Ashaveena Perumandla. Specifying and Checking Method Call Sequences in JML. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the International Conference on Software Engineering Research and Practice*, volume 2, pages 511–516. CSREA Press, 2005.
- 22 David R. Cok. JML and OpenJML for Java 16. In *FTfJP 2021: 23rd ACM International Workshop on Formal Techniques for Java-like Programs, 2021, Proceedings*, pages 65–67. ACM, 2021. doi:10.1145/3464971.3468417.
- 23 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.
- 24 Robert DeLine and Manuel Fähndrich. Typestates for Objects. In *18th European Conference on Object-Oriented Programming, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004. doi:10.1007/978-3-540-24851-4_21.
- 25 José Duarte and António Ravara. Retrofitting Typestates into Rust. In *25th Brazilian Symposium on Programming Languages*, pages 83–91. ACM, 2021. doi:10.1145/3475061.3475082.
- 26 Catarina Gamboa, Paulo Alexandre Santos, Christopher Steven Timperley, and Alcides Fonseca. User-driven Design and Evaluation of Liquid Types in Java. *CoRR*, abs/2110.05444, 2021. arXiv:2110.05444.
- 27 David Harel. Dynamic logic. In *Handbook of philosophical logic*, pages 497–604. Springer, 1984.
- 28 David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987. doi:10.1016/0167-6423(87)90035-9.
- 29 Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. A Tutorial on Verifying LinkedList Using KeY. In *Deductive Software Verification: Future Perspectives – Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, pages 221–245. Springer, 2020. doi:10.1007/978-3-030-64354-6_9.
- 30 Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. doi:10.1145/363235.363259.
- 31 J.P. Hollander. Verification of a model checking algorithm in VerCors, August 2021. URL: <http://essay.utwente.nl/88268/>.
- 32 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of Programming Languages and Systems*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.
- 33 Marieke Huisman and Raúl E. Monti. On the Industrial Application of Critical Software Verification with VerCors. In *Proceedings of Leveraging Applications of Formal Methods*, volume 12478 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2020. doi:10.1007/978-3-030-61467-6_18.

- 34 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniérou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.
- 35 Reiner Hähnle. Private communication, July 2022.
- 36 Bart Jacobs. Modular Verification of Liveness Properties of the I/O Behavior of Imperative Programs. In *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles, Proceedings*, volume 12476 of *Lecture Notes in Computer Science*, pages 509–524. Springer, 2020. doi:10.1007/978-3-030-61362-4_29.
- 37 Bart Jacobs. Private communication, March 2022.
- 38 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods – Third International Symposium, Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011. doi:10.1007/978-3-642-20398-5_4.
- 39 Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems – 8th Asian Symposium, Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer, 2010. doi:10.1007/978-3-642-17164-2_21.
- 40 Ioannis T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006. doi:10.1007/11813040_19.
- 41 Taekgoo Kim, Kevin Bierhoff, Jonathan Aldrich, and Sungwon Kang. Typestate protocol specification in JML. In *Proceedings of the 8th International Workshop on Specification and Verification of Component-Based Systems*, pages 11–18. ACM, 2009. doi:10.1145/1596486.1596490.
- 42 Sophie Lathouwers and Marieke Huisman. Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers. In *10th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2022*, pages 69–79. ACM, 2022. doi:10.1145/3524482.3527652.
- 43 Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006. doi:10.1145/1127878.1127884.
- 44 K Rustan M Leino and Peter Müller. A basis for verifying multi-threaded programs. In *European Symposium on Programming*, pages 378–393. Springer, 2009. doi:10.1007/978-3-642-00590-9_27.
- 45 K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009. doi:10.1007/978-3-642-03829-7_7.
- 46 Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992. doi:10.1109/2.161279.
- 47 João Mota, Marco Giunti, and António Ravara. Java Typestate Checker. In *Proc. of Coordination Models and Languages (COORDINATION)*, volume 12717 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2021. doi:10.1007/978-3-030-78142-2_8.
- 48 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of Verification, Model Checking, and Abstract Interpretation*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.
- 49 Oscar Nierstrasz. Regular types for active objects. *ACM sigplan Notices*, 28(10):1–15, 1993.
- 50 Peter O’Hearn. Resources, concurrency, and local reasoning. *Theoretical computer science*, 375(1-3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.

- 51 Peter O’Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.
- 52 Wytse Hendrikus Marinus Oortwijn. *Deductive techniques for model-based concurrency verification*. PhD thesis, University of Twente, 2019.
- 53 Matthew J. Parkinson and Alexander J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In *Proceedings of Programming Languages and Systems*, volume 6602 of *Lecture Notes in Computer Science*, pages 439–458. Springer, 2011. doi:10.1007/978-3-642-19718-5_23.
- 54 Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems, Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 158–182. Springer, 2015. doi:10.1007/978-3-662-46669-8_7.
- 55 John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. doi:10.1109/lics.2002.1029817.
- 56 R.B. Rubbens. Improving Support for Java Exceptions and Inheritance in VerCors. Master’s thesis, University of Twente, 2020. URL: <http://essay.utwente.nl/81338/>.
- 57 Ayesha Sadiq, Yuan-Fang Li, and Sea Ling. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software*, 159, 2020. doi:10.1016/j.jss.2019.110450.
- 58 Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An Automatic Verifier for Java-Like Programs Based on Dynamic Frames. In *Fundamental Approaches to Software Engineering, 11th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, Proceedings*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008. doi:10.1007/978-3-540-78743-3_19.
- 59 Robert E. Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929.
- 60 André Trindade, João Mota, and António Ravara. Typestates to Automata and back: a tool. In *Proceedings 13th Interaction and Concurrency Experience, ICE 2020*, volume 324 of *EPTCS*, pages 25–42, 2020. doi:10.4204/EPTCS.324.4.
- 61 André Trindade, João Mota, and António Ravara. Typestate Editor. <https://typestate-editor.github.io/>, 2022.
- 62 Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: Separating Permissions from Data in Rust. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473597.

The Dolorem Pattern: Growing a Language Through Compile-Time Function Execution

Simon Henniger ✉

Technische Universität München, Germany

Nada Amin ✉

Harvard University, Cambridge, MA, USA

Abstract

Programming languages are often designed as static, monolithic units. As a result, they are inflexible. We show a new mechanism of programming language design that allows to more flexible languages: by using compile-time function execution and metaprogramming, we implement a language mostly in itself. Our approach is usable for creating feature-rich, yet low-overhead system programming languages. We illustrate it on two systems, one that lowers to C and one that lowers to LLVM.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Language features

Keywords and phrases extensible languages, meta programming, macros, program generation, compilation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.41

Category Pearl/Brave New Idea

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.13>

Software: <https://zenodo.org/record/7720029>

Acknowledgements We thank Michael Ballantyne, Will Byrd, Anastasiya Kravchuk-Kirilyuk, and Cameron Wong for discussions about this work and feedback on drafts. We also thank anonymous reviewers for the insights and feedback.

1 Introduction

1.1 Motivation

Traditional macros serve mostly to create new syntax forms, expanding into a pre-defined core language. This means that the expressive power of macros is ultimately limited by the core language. For example, traditional macro systems could not add a “plus” macro to a language that does not have a concept of arithmetics.

This means that a language could never be fully built up from scratch with one of these macro systems. It also imposes a limit on the flexibility of macro systems: for example, they typically could not allow for supporting new features of a CPU chipset or a target language that are unsupported in the compiler.

We show a new pattern to design languages that can grow beyond their original definition. Our macros do not expand into a pre-defined core language, but rather into the target language itself. This allows us to build up a language of flexible zero- and low-cost abstractions in the form of macros.



© Simon Henniger and Nada Amin;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 41; pp. 41:1–41:27



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1.2 Idea

We start with a low-level system (*the target system*), think the C environment. Using this system, we create a minimal programming language that compiles to the target system and allows for compile-time function execution. This language (which we will later call the *base language*), is barely any more high-level than the target system and just big enough to call and define functions.

We then give this language access to its own code generation functions. We also allow for certain functions defined in the language to be executed *as part of the compilation*.

As a result, we can define entirely new language features in the language and immediately use them after defining them, thereby *bootstrapping* a language from a low-level system.

This has a number of implications. First, we no longer implement a compiler as a monolithic unit. Rather, ours is an extensible system that consists of the implementation of the base language and any additional language features implemented in the base language.

Second, we effectively have heterogeneous staging from the high-level that targets the low-level system. The staging is realized by executing compiled functions at compile-time.

Our pattern provides abstractions within the high-level system to define new language features. Those are based on traditional macros, which allow adding commands to the language. In addition to macros, we allow for layers. A layer is a set of changes to the behavior of a set of existing macros, i.e. a change to the language semantics. Layers give rise to a language tower.

We show that our language provides facilities for very flexible metaprogramming and exhibits minimal overhead in many cases.

1.3 Structure

In this paper:

- We describe our design principles (Section 2).
- We describe the related concepts of heterogeneous staging and compile-time function execution (Section 3).
- Based on these design principles, we implement a small demonstration system that targets C, called *dolorem-c* (Section 4).
- We evaluate and demonstrate the flexibility by showing that we can implement examples (Section 5).
- Based on the result of our evaluation, we design, implement and evaluate a larger system called *dolorem-llvm* (Section 6), which we also evaluate (Section 7).

We discuss related work (Section 8) and conclude (Section 9).

2 Design

2.1 Definition

A language uses the *Dolorem pattern* if:

1. It provides the `lower` form to lower high-level source code to a low-level target language.
2. It provides the language user with the ability to define *macros*, which change the behavior of the `lower` form.
3. It exposes internal code generation functionality of the compiler to macros. Macro code can therefore explicitly control code generation, as though it were part of the compiler.

4. Its macros are defined in exactly the same language as any other code, rather than in a special language only available at compile-time. In particular, macros can call into regular code, allowing for compile-time function execution.
5. It is constructed through bootstrapping and staging.

We call a language a *Dolorem language* if it uses the Dolorem pattern.

Finally, we call a system that reads and interprets or compiles programs written in a Dolorem language a *Dolorem system*.

2.2 Two languages

This paper shows the design of two Dolorem languages. One of them, `dolorem-c`, lowers to C code, while the other lowers to LLVM IR.

2.3 Design Goals

The five parts of the Dolorem pattern definition will become primary design goals of the languages we will show. We now further elaborate on them and will then describe a set of secondary design goals that follow from the definition and help us reach our primary design goals.

2.3.1 Lower, don't evaluate

`eval` is commonly a function that takes an expression, evaluates it immediately, and returns another expression: the result of the evaluation. For example, the result of evaluating `(+ 1 1)` would be `2`, and the result of evaluating `(car '(a b c))` would be `'a`.

Dolorem systems call `lower` instead of `eval`.

`lower` takes an S-expression, parses it, lowers it into the system's target language, and returns a reference to the lowered code. The reference could be a pointer to an SSA in LLVM's Intermediate Representation, or just a string with C code. Calling `lower` on a function call, a variable name or an integer never actually calls the function, loads the variable content, or returns the value of that integer – with one exception: macros.

2.3.2 Macros are special cases of lowering

When the `lower` form is called on something that looks like a function call, it checks if the called function is marked as a macro.

If it is, it delegates to the macro which means it immediately (i.e. during compile time) calls it on its own argument and returns its result. This makes macros an example of compile-time function execution (see 3.1).

Hence, macro usages are one (and the only) instance in which the `lower` form behaves similar to the `eval` form.

Because macros have access to the entire language, the calling behavior is the only difference between macros and functions. Apart from it, they are both treated the same. This includes being compiled the same way.

2.3.3 Give macros explicit control over code generation

In Dolorem, macros can generate their own code. In order to do this, we give them explicit access to the code generation facilities in the compiler, so they can explicitly emit the target language.

2.3.4 Give macros access to the entire language

Unlike in other systems like C++'s “constexpr” functions, macros should not be restricted in which other functions they can call and what they can do.

First, this allows us to use macros in traditional metaprogramming cases, for example those where a metaprogram accesses and parses a file to use it as a basis for generating code. Second, since we use macros to directly generate backend code, it also improves the overall flexibility of the language.

Note that this requires us to consider each individual function to be its own translation unit. If we delayed compilation of a function, there could be a macro usage between the definition and compilation of it and then the function could not be used by the macro.

2.3.5 Keep the base language as small as possible

Because the Dolorem pattern uses staging, a Dolorem language is implemented in two parts: One part is implemented in some other language like C, while the rest is implemented in the Dolorem language itself (on top of the base language).

We refer to the first part as *the base language*. This part of the system should be kept minimal, i.e. just big enough to allow to bootstrap the rest of the language.

First, the Dolorem pattern aims for low overhead, so there is usually no reason not to implement something in a Dolorem language if we can.

Second, while we work to minimize it (see below), there is and will always be a barrier between a Dolorem language and its target language. So the behavior of any code that is written in a target language is a bit harder to change from within the Dolorem language.

2.4 Secondary principles

Based on this definition of the Dolorem pattern, we define a few secondary design goals that help us reach our primary goals:

2.4.1 Limit barriers

There should be as few barriers between the Dolorem language and the target environment as possible. For example, we should allow one to call the other directly, without using a *foreign function interface* or worrying too much about name mangling.

This not only makes it easier to interface Dolorem programs with existing programs written in the target environment, e.g. C (thereby increasing Dolorem's value as a systems programming language), but also allows to access the compiler's backend as seamlessly as possible.

2.4.2 Do not prescribe a model of execution

We let the language user decide about when to write code to disk/execute it rather than prescribing one model of execution. Unlike traditional language processors, Dolorem systems do not have a preferred mode of translation. Although we like to refer to them as compilers, they are neither compilers nor interpreters in the traditional sense.

Rather, a language user can decide to write and use a macro to execute code at compile-time, thereby using the Dolorem system as an interpreter, or write and use a macro to write generated code to disk, thereby using it as a compiler.

2.4.3 Layers are sets of overridden macros

We want to be able to change the behavior of macros even after their definition.

That is why we allow macros to be overridden. When we override a macro, we provide a new macro that replaces it. This new macro may call into the original macro (or an older override) as a fallback.

Since it is usually more useful to override several macros at once, Dolorem systems support layers – sets of macros that are overridden together. Layers are often used to provide new language features. For example, we will later show a “function overloading” layer that overrides the `lower` macro and the `defun` macro to allow for several functions with the same name and different numbers of arguments.

Since most macro overrides fall back on the last override, layers can stack to form a tower.

2.4.4 At global scope, everything is a macro

Similar to Lisp, there are no operators or keywords that are not also macros. This means that, at global scope, a Dolorem program is only a sequence of macros that are executed by the compiler in order of appearance.

Having nothing but macros is useful because it allows us to change every aspect of the language by overriding macros.

2.4.5 Use S-expressions

This paper is not about syntax. That said, the Dolorem pattern does place some unusual constraints on the syntax of the language it is used in:

1. The language itself is defined to be as flexible as possible and we do not want syntax to be a limiting factor.
2. We want macros to be able to read the syntax tree. Hence, the base language must provide facilities to do so. Any complexities in the syntax will therefore bloat the base language.

S-expressions are both flexible, and very easy to read and manipulate programmatically (using the famous `car`, `cdr`, `cons`), making them an ideal fit.

3 Concepts

3.1 Compile-time function execution

Compile-time function execution refers to a compiler’s ability to execute functions written in its target language at compile time.

Often, this is done to compute constant expressions at compile-time. An example is C++’s `constexpr` [2].

Depending on the language, functions that can be executed at compile time may only have access to a subset of the language. In C++, for example, `constexpr` functions are relatively limited: they can only call other functions marked `constexpr` and can not access the environment.

As explained above, languages with the Dolorem pattern use compile-time function execution to bootstrap the language. Macros, the functions executed at compile time, have access to the entire language. This includes the ability to call non-macro functions (principle (2.3.4)).

3.2 Heterogeneous staging

Multi-stage programming offers a principled way of generating code by working in a language with two stages: the first stage is a program generator which when executed yields the second-stage program. Whether a program belongs to the first stage (code generator) or second stage (generated code) can be determined by syntax quotations (MetaML [10]) or types (LMS [9]). Some multi-stage programming systems like LMS (Lightweight Modular Staging) support heterogeneous staging, where the code generator and the generated code can be in different languages.

Dolorem can be seen as a form of heterogeneous staging, where the `lower` facility acts as staging. Which stage a function belongs to is determined by whether it is marked as macro. Macros, being executed during code generation, form the code generator stage, while regular functions are the second stage.

4 dolorem-c: Implementation with C Target

dolorem-c is our first attempt at an implementation of the Dolorem pattern for a system. dolorem-c is intended as a proof of concept and simple demonstration. We want to distill the essence of the Dolorem pattern, rather than present a fully viable language¹.

To keep it as simple as possible, we target C and leave out some aspects of the language for now – for example, we rely on C for the type system and do not implement any type checks ourselves.

Since the dolorem-c system has no dependencies except for a working C compiler, it is easy to try it out².

4.1 The `lower` macro

After reading a file, dolorem-c calls `lower` on its content. `lower` takes a list and returns generated code. In our case, this means it has the following function signature:

`struct cexp* lower(struct val* e);`, where `struct cexp*` is a type that contains a piece of generated C code and `struct val*` is an S-expression.

Any C transpiler needs to implement its own representation of C code. The primary constraint for ours is that we want the resulting interface to be as simple as possible to use. That is why we keep our representation of a value in C code to only four syntactic categories (each stored as a separate string) that we use to track whether a statement should be at global or local scope, and to be able to differentiate between expressions and full statements that precede them. These are the four strings:

- `expression`, which stores the expression itself, e.g. `a+b`. It is later inserted into a statement.
- `context`, which stores a list of statements that need to precede whatever statement `expression` will be inserted into, e.g. `int a;`. If it is non-empty, this always ends in a closing curly parenthesis or a semicolon.
- `global`, which is a list of statements on global scope that should precede whatever function this `cexp` will be inserted into.
- `header`, which is a list of headers at global scope that need to be added to the environment after `cexp` has been compiled, e.g. `#include <stdio.h>` or a `cexp` that contains `printf`.

¹ The entire dolorem-c source code can be found in this GitHub repository: <https://github.com/metareflexion/dolorem-c>

² A version that runs in the browser can be found here: <https://shenniger.github.io/try-dolorem-c>

In the following, we will write `cexps` as tuples of strings in square braces separated by vertical lines, e.g. `[a+sqrt(b)||#include <math.h>]`.

For illustration, this is the behavior of `lower`:

- If `lower` is called on a number or string literal, it can directly lower that literal to C and prints that number or string literal as a `cexp` with empty local and global context (i.e. `4` evaluates to `[4|||]`).
- If `lower` is called on an identifier, it assumes it has found a reference to a variable and returns that as a `cexp` with empty local and global context (i.e. `myvariable` evaluates to `[myvariable|||]`). A more advanced Dolorem system (like `dolorem-llvm` which will be shown later) would try to find the variable in the current scope. `dolorem-c` leaves this to the C compiler.
- If `lower` finds a function call, it checks if the called function is a macro. If not, all the arguments to the call are recursively `lowered`, and code for the call is generated. If the called function is a macro, `lower` resolves the address of the macro function, and calls it.

4.2 Macros in the base language

We want to bootstrap as much of the language as possible. In order to be able to do that, we need to add a few more macros, just enough for the user to be able to define their own functions and macros:

- `progn` calls `lower` on each of a list of expressions and returns the result of the last (similar to `progn` in Lisp).
- `include` reads another file, and calls `progn` on its content.
- `funproto` creates a function prototype for a given function signature.
- `function` creates a function (including body).
- `mark-as-macro` marks a given function name as a macro.
- `compile` lowers its argument, then compiles the resulting C code to machine code, and stores the C headers. See below for more.

Note that `mark-as-macro` and `compile` are the only two macros in this list to have side effects. The others do not modify global state at all – for example, `function` only returns C code with a function definition, but does not register that function anywhere yet.

We need `progn` and `include` in the base language because, without them, we could not read any files of source code (but only individual macro calls), and we need the others to be able to define functions and macros.

In addition to those macros, there are a few functions that help with reading and processing S-expressions like `car`, `cdr`, `val-is-string`, `expect-ident`, or `make-int-val`.

The base language contains nothing else – most notably absent are any kind of control structures, local or global variable definitions, and arithmetic expressions.

In order to reduce barriers, the base language also does not introduce any new calling conventions. One example of that is that `dolorem-c`'s calling convention is the same as that of C and all `dolorem-c` symbols need to exist within the C environment. This is even true for macros: When a macro is called within `lower`, the address of the macro is being resolved by using a simple `dlsym(RTLD_DEFAULT, "macroname")`, i.e. we use the system's dynamic linker to resolve the symbol.

4.3 Marking a function as a macro

Any function that has the macro signature (`struct cexp* myfunction(struct val* e);`) can be marked as a macro. This does not change anything about the function's code or calling convention on the C side, however it has implications on how calls to the function are handled in `dolorem-c` code.

41:8 The Dolorem Pattern

If `myfunction` is not marked as a macro, `(myfunction myarg)` generates code to call the function and evaluate its argument `myarg`. Since we are assuming `myfunction` to take only one argument, a `struct val*`, `myarg` would thus be expected to be a pointer to a list argument.

If `myfunction` is marked as a macro, `(myfunction myarg)` resolves the macro `myfunction` and calls it on the unevaluated S-expression form of the argument (i.e. a list of only one element: the string `myarg`).

Note that macros can be overridden, functions cannot. Thus, a call to a function is resolved by the C environment, while a call to a macro is resolved first by Dolorem's own macro table (which might say that the macro `myfunction` was overridden by the macro `myfunction2`, so that macro would be called instead).

The `lower` function has the same signature as macros, and we also mark it as a macro. This may seem counterintuitive at first, because there is no point to ever calling it as a macro (as anything in `dolorem-c` is evaluated by `lower` anyway). Marking `lower` as a macro is still necessary, because that way, we can override it.

It is, however, useful to be able to call `lower` as a function from within `dolorem-c` code. For that, we define a function `lower-now`, which acts as an adaptor and calls the current `lower` override.

4.4 Implementing base language macros

Base language macros are implemented as simple C functions. As an example for how C code interfaces with S-expressions and macros, here is `progn`:

```
struct cexp *progn(struct val *e) {
    struct cexp *r = make_cexp("", "", "", "");
    // iterate through list
    for (struct val *args = e; !is_nil(args); args = cdr(args)) {
        // call lower on each element
        struct cexp *a = call_macro("lower", car(args));
        add_cexp(r, a);
        // if not the last element, add
        // expression to context
        if (a->E && *a->E && !is_nil(cdr(args))) {
            appendString(&r->Context, print_to_mem("%s;\n", a->E));
        }
        r->E = a->E;
    }
    return r;
}
```

We begin by calling `make_cexp` to create an empty C expression. Note that the four empty strings we pass to it correspond to the four syntactic categories described above.

Then we iterate through the list we were passed, lowering each element, and adding its context, global, and header to our `cexp`. The expression of the `cexp` we return will be that of the last element of the list (because that is the return value of the block); any expressions of the preceding elements will be appended to `context`.

Note that we do not directly call `lower`, but rather write `call_macro("lower", ...)`. While there is a C function called `lower` that we could use, the macro `lower` might have been overridden by Dolorem code by this point, so we need to look up the current version of `lower`.

4.5 Writing functions in the base language

In the base language, simply writing `(function ...)` is not enough to define a new function. That is because `function` merely generates the `cexp` for a function, but never actually invokes a C compiler.

Actually, we need to wrap any function definition into a `compile`, for example: `(compile (function hello-world ()void (puts "hello, world!")))`

The `compile` macro first `lowers` its argument, then appends the header portion of it to a global header store, and invokes a C compiler³ to compile it to a shared library⁴. The `compile` macro also loads this shared library.

4.6 Example macro 1: `defun`

To reduce this boilerplate, we first define `defun`, which provides us with a nicer function definition syntax and makes compilation implicit. `defun` is a good example for a simple macro that does not explicitly generate C code, but rather uses existing compiler facilities, for which it generates new input Dolorem code.

We call this kind of macro (that transforms Dolorem code to Dolorem code and then calls the `lower` macro on it for code generation) a *homogeneous transformation* (see 4.9).

In order to define a macro, we define a function that takes an S-expression and returns a `cexp`. The dolorem-c type system cannot handle either at this early stage of bootstrapping, so the actual function signature is `void* defun(void* args);`. We compile the resulting function, and then mark it as a macro.

`defun` should be just like `function`, except that it automatically compiles for us. Since all we want is this small syntactic change, it makes sense to implement the macro as a homogeneous transformation (from Dolorem code to Dolorem code):

```
(compile (function defun ((void* args)) void*
  (call-macro "lower" (cons
    (make-ident-val "compile")
    (cons (cons (make-ident-val "function") args) (make-nil-val))))
  ))
(mark-as-macro defun)
```

To define a macro, we first create a function, then explicitly compile it, and finally mark our function as a macro. Within the function, we create a new list (using standard Lisp macros like `cons`) composed of a call to `compile` and `function` on our argument.

We then `lower` this list. Note that, rather than directly calling `lower` in the macro, we need to `call-macro "lower"`. We need to do so because `lower` is a macro, but in this case, we do not want to call it during code generation of `defun`, but rather want to add a call to `lower`. We could also call `lower-now` (see 4.3) and will do so in later code examples.

After `defun`, we bootstrap a `defmacro` in a similar way, such that we no longer need to write down the function signature and explicitly `mark-as-macro` every time. Note that, because of the way in which we define `defmacro`, it automatically creates a parameter called `args` that contains the list the macro was called on.

³ Currently, both `tcc` and `gcc` have been tested. `tcc` is preferred on Linux because of its much faster compilation speed.

⁴ We use a shared library to make sure `libdl` can find all new symbols, and that newly compiled code can find and access all the symbols defined before. This is why the C compiler is asked to link the new code into a shared library (with dynamic symbol lookup enabled), which is then immediately loaded. `dolorem-llvm` uses a more complicated JIT-based approach for this, which we will discuss later.

4.7 Example macro 2: `var`

We will now show `var` as an example of a macro that generates its own C code.

These macros, which we will refer to as *lowering transformations* lower their Dolorem input to C code.

In the case of the `var` macro, we tap into C's ability to define local variables. We want to be able to write `(var int x)` to create a new int variable called `x`. For this, we define a primitive macro called `var`:

```
(defmacro var
  (make-cexp (expect-ident (car args))
    (print-to-mem
      "%s %s;\n"
      (expect-type (car (cdr args)))
      (expect-ident (car args)))
    ""
    ""))
```

We use the newly defined `defmacro` to define the macro in only one step (rather than defining a function, compiling it, and then marking it as a macro). Within the macro's body, we call `make-cexp`, whose four arguments correspond to the four syntactic categories defined above. Two of them (`header`, `global`) are empty, while the first one is simply the name of the variable, and the second is C syntax for its definition.

The base language gives us a range of functions like `char* expect_ident(struct val* e)`; and `long long expect_int(struct val* e)`; that check whether a given list expression is of a certain type and, if so, unpack and return its value (otherwise, the function outputs an appropriate error).

The macro uses `print_to_mem`⁵, a C function defined in the compiler that acts like `asprintf` (i.e. it formats a string and automatically allocates memory for it) to transform the dolorem-c code into C code and creates a `cexp` with the variable definition as context, only the variable name as expression, and empty `global` and `header` values.

In a similar way, we bootstrap (among others) `cond`, a Go-like `:=`, `and`, `equals`, `loop`, `add`, `sub`, and `assign`.

4.8 Language usage example: Hello, world in a loop

With the very primitive base language and our macros, we can now implement this example that prints "Hello, world!" ten times:

```
(include "def.dlr")
(defun main () void (progn
  (:= count 0)
  (while (sub count 10) (progn
    (puts "Hello, world!")
    (assign count (add count 1)))))
```

We assume that the file `"def.dlr"` contains all those macros and a function prototype for `puts`.

⁵ As a purely cosmetic improvement, any time an underscore appears in the C code, we can write a dash in dolorem-c. All dashes in function names are replaced by underscores when they are read.

Note that, while our program defines a `main` function, it currently does not do anything. As explained above, in order to execute code, we need to write a macro that either directly calls into the code, or writes it to disk.

We will show how to use Dolorem systems as compilers later. For now, let's execute our code directly:

```
(defmacro run (progn
  (main)
  (make-cexp "" "" "" ""))
(run))
```

We create a macro called `run` that calls into `main` and returns an empty `cexp`. Then, immediately after defining the macro, we call it.

4.9 Different types of transformations

As discussed above, there are two archetypes of Dolorem macros:

- *Homogeneous transformations*: These macros transform the S-expression they are given and then (at their very end) call `lower` to generate code for the transformed expression. Lisp can only work with this kind of macro.
- *Lowering transformations*: These macros parse the S-expression they are given and then call into the backend to generate code for it.

To demonstrate the difference, we will show two different implementations of the `quote` macro.

Most Lisps include a `quote` macro that, rather than evaluating an expression, passes its original S-expression along. Quoting is equally useful in Dolorem, but must be implemented differently. Rather than simply returning the original S-expression, we must generate code to create a given S-expression.

First, we will implement `quote` as a homogeneous transformation. We start with a function that, given a list, returns, as a list expression, a sequence of function calls to generate that list. For example, for `(1 2 3 test)`, our function should return `(cons (make-int-val 1)(cons (make-int-val 2)(cons (make-int-val 3)(cons (make-string-val "test")(make-nil-val))))))`. We call this function `lower-quote`. We can define the actual macro as simply `(defmacro quote (call-macro "lower" (lower-quote args)))` (i.e. call `lower-quote`, lower the result and return it).

`lower-quote` is defined as a series of `cond` statements⁶ that call a number of simple helper functions like `make-make-funcall` or `make-cons-funcall` that generate the actual calls. This is what the statements for lists and integers look like:

```
(defmacro quote (progn
  (var res void*)
  (cond (val-is-int args)
        (assign res (cons (make-ident-val "make-int-val") (cons args (
          make-nil-val))))))
  # ...
  (call-macro "lower" res)))
```

⁶ `dolorem-llvm` allows the implementation of a proper three-way if. In `dolorem-c`, this is hard due to the missing type system.

41:12 The Dolorem Pattern

This has several advantages. First, it is relatively easy to read (and, in fact, after `quote` has been defined, such functions are even more readable). It also does not require much advanced knowledge on the inner workings of the compiler or C. Furthermore, this macro is relatively forward-compatible and will continue to work with updated code generators, although future Dolorem syntax changes may break its compatibility.

Its primary issue is that it is not particularly fast. Creating a list expression that is then immediately parsed by the `lower` macro called in the last line is wasteful. We can write a faster version of it as a lowering transformation that directly generates C code:

```
(defmacro quote (progn
  (var res char*)
  (cond (val-is-int args)
        (assign res
                 (make-cexp (print-to-mem "make_int_val(%i)" (expect-int args))
                            "" "" "")))
  r))
```

While it is lower overhead, this way of implementation is also much more error-prone as we have to directly manipulate strings of C code.

There is no universal rule for when to use which of the two types of transformations. Both of them have their own advantages and disadvantages: lowering transformations are faster and more flexible, but error-prone, while homogeneous transformations are simpler, yet slower.

4.10 Example macro 3: Arithmetic operators

One of the traditionally more repetitive tasks within writing a compiler is to write the code for all the arithmetic operators. It is almost always the same code for each operator, which leads to code duplication within the compiler. We will show how to leverage dolorem-c's metaprogramming abilities to help us with bootstrapping the language and avoid having to write each operator separately.

An arithmetic operator needs to read the left-hand side and the right-hand side expression, then append the context of one to the other, and finally set the expression to something like "a+b". In order to save memory, we do not create a new expression value and instead reuse one of the arguments. Here is an example for `add`:

```
(defmacro add (progn
  (:= left (lower-now (car args)))
  (:= right (lower-now (car (cdr args))))
  (add-cexp left right)
  (set-expression left (print-to-mem "(%s) + (%s)" (get-expression
    left) (get-expression right)))
  left))
```

We see that almost all of this will be the same for all operators, apart from the name of the macro and the format string ("`(%s) + (%s)`").

Using a `quasiquote` implementation based on the `quote` macro shown above, we quote the entire macro, adding in the two changing parts via `quasiunquote`:

```
(quasiquote (defmacro (quasiunquote (car args)) (progn
  (:= left (lower-now (car args)))
  (:= right (lower-now (car (cdr args))))
  (add-cexp left right)
```

```
(set-expression left (print-to-mem (quasiunquote (car (cdr args)))
  (get-expression left) (get-expression right)))
left)))
```

Finally, we wrap this into a call to `lower` and add it into a macro:

```
(defmacro generate-arithmetic-operator
  (call-macro "lower" (quasiquote defmacro (quasiunquote (car args)) (
    progn
      (:= left (lower-now (car args)))
      (:= right (lower-now (car (cdr args))))
      (add-cexp left right)
      (set-expression left (print-to-mem (quasiunquote (car (cdr args)))
        ) (get-expression left) (get-expression right)))
    left))))
```

We have now defined a macro that defines macros.

To use it, we write the following (at global scope):

```
(generate-arithmetic-operator add "(%s) + (%s)")
(generate-arithmetic-operator sub "(%s) - (%s)")
(generate-arithmetic-operator mul "(%s) * (%s)")
(generate-arithmetic-operator divi "(%s) / (%s)")
```

4.11 Macro Overriding

With what we have shown so far, plenty of macros can be implemented into `dolorem-c`. However, there is currently no way to change a macro once it is defined.

Macro overriding changes that. We add a virtual table of macros to the base language. Now, whenever a function is marked as a macro, its address is stored in the virtual table, and whenever we call a macro, we take the address from the table rather than asking the dynamic linker for it.

We also define a new function `macrofunptr override_macro(const char* name, const char* newfun)`; that overrides the macro `name` by the replacement function `newfun` and returns a pointer to the old entry in the table.

With this, we can use the standard C pattern to change the behavior of existing macros:

1. We store the old function pointer from the virtual table in a global variable.
2. Then, we override the virtual table entry with a new function.
3. In the new function, we use the old pointer in the global variable as fallback.

4.12 Overriding Example: Location hints

A useful easy example is this overriding of `lower` that adds location hints to the C code so that the C compiler can emit correct debug information⁷. Any `dolorem-c` code compiled after a call to the function we are about to define (`load-csrchints`) can be stepped through line-by-line in GDB.

We start by creating a global variable to store the current `lower` function: `(compile (global-var csrchints_old_lower macrofunptr))`

⁷ gcc supports reading hint lines in the format of “# <filename> <line number>” to be able to tell where a piece of source code is originally from. We add these to the ‘context’ of the `cexp`.

41:14 The Dolorem Pattern

After this, we define a macro called `load-csrchints` that loads the C source hints override. This is where the overloading happens. The macro is supposed to be used at global scope, so we make it return an empty `cexp`:

```
(defmacro load-csrchints (progn
  (assign csrchints-old-lower
    (override-macro "lower" "lower-csrchints")))
  (make-cexp "" "" "" ""))
```

Finally, we define the new macro `lower-csrchints` that will replace `lower`:

```
(defmacro lower-csrchints (progn
  # Call lower-level 'lower'.
  (:= r (csrchints-old-lower args))
  (var filename char*)
  (var line long)
  (get-loc-info args (ptr-to filename) (ptr-to line) 0 0)
  (append-cexp r
    (get-expression r)
    (print-to-mem "# %li \"%s\"\n" line filename) "" ""))
  r))
```

Note that this implementation prints duplicated hint lines whenever many subexpressions are on the same C source line. It also currently does not handle blocks correctly, often giving only the number of the first line of a multi-line block. Although it is not entirely correct, it is good enough to be useful in many debugging scenarios.

4.13 Layers

As explained above, we want to be able to override multiple macros at once and use a convenient syntax that abstracts all the details (global variables, calls to the `override-macro` function, etc).

For example, we want to be able to write `(new-layer foo (lower (new-lower-code...))`) to override `lower` and to have all overrides, variables, and a function `load-layer-foo` automatically generated.

In `dolorem-c`, this higher-level layer syntax is defined as a homogeneous transformation macro, and it is not part of the base language, but rather defined in the language itself.

4.14 Layer Example: Function overloading

To demonstrate layers, we will add a very basic version of function overloading to `dolorem-c`.

`dolorem-c` does not have a type system, therefore function overloading by type is not possible. Hence, this example only overloads on the number of arguments, not the type.

To implement function overloading, we need to do two things. First, we need to change the `defun` macro such that it mangles the name of any newly created function name by adding the amount of arguments to the name. Second, we change `lower` such that it correctly resolves any overloaded function calls.

Note that one of the rather unusual features of this overloading layer is that we want mangled and non-mangled functions to coexist as seamlessly as possible. This is necessary because, otherwise, loading the layer would interfere with calling any functions defined before it was loaded. In order to achieve this goal, we add a hashmap that contains the names of any mangled functions. Whenever we find a function whose name is not in the hashmap, we do not change the call.

We will only change `defun`, not `function`. This means any functions that are defined using other means will not be mangled. This is useful because it avoids breaking macros or any other mechanism that might be defined and needs more low-level control over functions.

This is the entire source code of the layer:

```
(compile (global-var overloaded-fun void*)) # hashmap for names of
  mangled functions
(new-layer funoverload
  (init (assign overloaded-fun (hashmap-new)))
  (defun (progn
    (:= name (car args))
    (:= n-args (count-len (car (cdr args))))
    (hashmap-put overloaded-fun (expect-ident name) "")
    (val-set-string name (print-to-mem "%s__fo_%i" (expect-ident name)
      ) n-args))
    (fallback args)))
(lower (progn
  (cond (val-is-list args) (progn
    (var dummy char*)
    (cond (not
      (hashmap-get overloaded-fun
        (expect-ident (car args))
        (ptr-to dummy)))
      (val-set-string
        (car args)
        (print-to-mem "%s__fo_%i"
          (expect-ident (car args))
          (count-len (cdr args)))))))
    (fallback args))))))
```

We can now load the layer with `(load-layer-funoverload)`.

Within the layer implementation, we first add some initialization code for the hashmap, and then override the `defun` macro. Our new `defun` looks for the name of the function to be defined, saves it in the hashmap, and then mangles it by appending `"__fo_X"` (where `X` is the amount of parameters). Finally, it calls whichever `defun` implementation is in the tower before this layer on the newly changed definition.

We also override `lower`. If it finds a function call, our new implementation checks if the function name is in the hashmap. If it is, it mangles it based on the amount of arguments supplied. Finally, it delegates to the next layer.

4.15 Implementing optimizations

We want to be able to write code that transforms and optimizes already-generated code. To do so, we implement a transformation layer, similar in principle to the one implemented in section 4.12 (which adds location hints), except that our new layer will change the code.

As we implement our optimization, we will run into one major limitation, which is not related to the concept of the Dolorem pattern, but rather to a specific design decision we made for `dolorem-c`. Our representation of C code is purely textual, as opposed to using a syntax tree that is easier to change, making it a pain to read and transform once it has been generated. Using another representation for C code (such as some kind of syntax tree) would have made this easier. Unfortunately our textual representation will make our optimization more inelegant and inflexible than necessary.

41:16 The Dolorem Pattern

In section 4.10, we added a division operator to `dolorem-c`. It is common for compilers to transform integer divisions of a variable and a constant to a bit shift if the constant is a power of two. We will implement this optimization (except, since we are only showing this as an example and try to simplify as much as possible, we will only transform a division with two to a bitshift and ignore the other powers of two)⁸.

If we want to transform division code, we can either (1) change the `divi` macro by overriding it in a layer, or (2) look for divisions in the code after it has been fully generated, and then change the code. Only (2) is a transformation in the narrow sense, as (1) does not technically transform generated code, but rather changes the code generator. In the Dolorem context, both have a similar effect.

We will first implement a layer for (1):

```
(new-layer bitshift-instead-of-division
  (divi (progn
    (:= first-operand (car args))
    (:= second-operand (car (cdr args)))
    (:= result NULL)
    (cond (and
      (val-is-int second-operand)
      (equals (expect-int second-operand) 2)) (progn
        (:= c (lower-now first-operand))
        (assign result (make-cexp
          (print-to-mem "(%s) >> 1" (get-expression c))
          (get-context c)
          (get-global c)
          (get-header c))))))
    (cond (not result)
      (assign result (fallback args)))
    result)))
```

This overrides the `divi` macro itself, and adds a check for whether one of the operands is the number two.

Alternatively, we can implement option (2) and search and replace existing code for any divisions with two. Due to the limitations with the C code representation, we do this on a textual level.

All divisions will be within functions, so the easiest way to catch all of them is to override the function macro (used within `defun` as described above) and work with its output:

```
(new-layer bitshift-instead-of-divisions-2
  (function (progn
    (:= code (fallback args))
    (:= global-ptr (get-global code))
    (:= pointer global-ptr)
    (:= modified 0)
    (loop (assign pointer (strstr pointer "/" (2))) (progn
      (memcpy pointer ">> 1" 5)
      (assign modified 1)))
    (cond modified
      (set-global code global-ptr))
    code)))
```

⁸ Since all code is compiled (and optimized) by a C compiler which will most certainly do this optimization, our code does not actually yield a speed boost, but merely serves to illustrate how code transformations are implemented with the Dolorem pattern.

First, we call the overridden macro to generate the function code and then we search the `global` of its result (which is where the function body will be).

Note that this second implementation does not refer to the `divi` macro at all. In fact, it does not matter whether the division was created by that macro, some other macro, or even a combination of several macros. All that matters is that the generated code eventually shows up in a function, and once it does show up in the function, we have access to all other code in the function and can use this contextual information for our optimization. That is how we can also implement more complex optimizations that touch multiple macros.

As we discussed, the transformations themselves are implemented in a rather inelegant way, but still, this experiment shows the principles of how source code transformations can be implemented in Dolorem:

1. For simple optimizations, it often makes sense to change how the code is being created, as opposed to transforming it after it was already created.
2. To transform code, create a layer around a macro like `function` (or `compile`) that all code will pass through, and modify it there. Then, our optimization can even touch multiple macros and operate on an entire function (or even several functions).
3. The specific range of practical transformations depends on the design of the data structure generated code is stored in. The Dolorem pattern itself is not a limiting factor.
4. If an optimization is impractical to do, that does not mean it is impossible entirely. Dolorem systems use a target language and target compiler outside of Dolorem. Sometimes, it can make sense to optimize there.

More experimentation is needed to see whether this approach scales to more complex optimizations.

5 Discussion of the C implementation

We have successfully implemented a Dolorem system that targets C.

5.1 Code size

Minus header files, the hashmap implementation, the driver and the reader/parser, the entire compiler only has 377 lines of code. This shows that we were able to keep the base language very small. In an additional 284 lines of Dolorem code, we were able to bootstrap a relatively usable language. This confirms our original plan to keep the base language minimal and bootstrap the rest.

5.2 Scope

We have left out some important aspects of a language, most importantly the type system. This is certainly a major reason for why the implementation is so small in terms of code size. In the rest of the paper, we will show that the Dolorem pattern can scale and work for a more complicated language.

5.3 Generated code

When we output C code generated by `dolorem-c` and change the formatting a bit, it looks very similar to human-written C code (see Figure 1). Of course, this depends on the macros used. Macros may introduce additional complexities, and thus, additional overhead, if the abstraction they provide is costly or if they are badly written and introduce unnecessary cost – but none of the macros we have shown or used in examples do.

41:18 The Dolorem Pattern

■ **Figure 1** *left*: dolorem-c code for a non-recursive Fibonacci function, *middle*: generated C code (indentations added for readability), *right*: hand-written C code.

```
(defun
 fib ((int x)) int
 (progn
  (var i int)
  (var n1 int)
  (var n2 int)
  (var n3 int)
  (loop
   (less i x)
   (progn
    (assign n3 (add
              n1 n2))
    (assign n1 n2)
    (assign n2 n3)
    (assign i (add i
                  1)))
  ))
 n3
))
```

```
int fib__fo_1(int x) {
  int i = 2;
  i;
  int n1 = 0;
  n1;
  int n2 = 1;
  n2;
  int n3 = 0;
  n3;
  while((i) < (x)) {
    (n3) = ((n1) + (n2));
    (n1) = (n2);
    (n2) = (n3);
    (i) = ((i) + (1));
  }
  return n3;
}
```

```
int fib(int x) {
  int i = 2;
  int n1 = 0;
  int n2 = 1;
  int n3 = 0;
  while (i < x) {
    n3 = (n1 + n2);
    n1 = n2;
    n2 = n3;
    ++i;
  }
  return n3;
}
```

This also means we can compile it with a traditional C compiler, and the binary will be basically indistinguishable from that created from an equivalent program originally written in C.

5.4 Run-time overhead

The above already gives us a first indication that dolorem-c has no or very low overhead. Since the implementation is more of a proof-of-concept, we have not measured its performance, but have rather, in many cases including the one shown in Figure 1, seen that its C code is essentially equivalent to what we would have written by hand. We will later show a more detailed run-time analysis for dolorem-llvm.

5.5 Compile speed

Since dolorem-c can be used to directly execute code (rather than as a transpiler which code is exported from), compile speed is essential.

We expect most of the overall compile time to be spent in the C compiler. In order to verify, we implement a command line flag (-M) that instructs the compiler to measure the overall run-time and the time spent waiting for calls to the compiler, and output a percentage.

We test this while compiling several programs examples, including the definitions of the macros presented here (and more). None of our test cases does anything, they all compile and then terminate.

As expected, measurements show that 95-97 % (tcc) or more than 99 %⁹ of overall compile time is spent in the C compiler. The discrepancy is due to the fact that gcc is about 20 times slower than tcc, likely because its code generation is generally slower and also because it is not optimized to be invoked for each function separately.

Subtracting the time spent in the C compiler, dolorem-c takes between 5 and 10 milliseconds to compile the entire collection of standard macros (around 500 lines of dolorem-c code)¹⁰.

These measurements show that, while the dolorem-c compiler is comparatively fast given its flexibility, C compilation drastically slows it down. To improve speed, the implementation should implement some form of caching for the compilation. We will later show an example of this for dolorem-llvm.

6 Implementation with LLVM Target

We are now ready to implement the Dolorem pattern in an LLVM-based [7] system. While dolorem-c was intended as a proof of concept and simple demonstration, dolorem-llvm is intended to include more complex features like static typing, even if this makes it more complex¹¹.

The design of dolorem-llvm is similar to dolorem-c in many ways. We will not provide a full outline of the design, but rather describe the differences to dolorem-c, with a focus on providing some general discussion on how to apply the Dolorem pattern to a more complete compiler system.

Rather than generating code as a string, dolorem-llvm calls into LLVM directly to build LLVM IR.

LLVM's main interface is template-heavy C++ code. Since interfacing with that would drastically increase the size of the base language, we instead mostly interface with the C bindings (llvm-c).

6.1 The `lower` macro

Obviously, the most important difference is that there are no `cexps` anymore, because the new system does not translate to C.

Instead, `lower` returns a pointer to an `rtv` (*run-time value*) rather than a pointer to a `cexp`. An `rtv` wraps an `LLVMValue`, i.e. a reference to a value computed by a program. It also contains the type information the new type system needs.

6.2 Functions

`LLVMValue` works differently from a `cexp` in one key way: Because a dolorem-llvm `rtv` can hold only a value within a function, rather than an arbitrary piece of code, a dolorem-llvm macro can not return a full uncompiled function.

⁹ The results are very similar for all test cases and, while there is quite some variance between runs, it seems random and unrelated to the length or any other property of the test case. That is why we do not provide a detailed list of measurement values.

¹⁰ Measured on an Intel Xeon E3-1505M v5, when compiling the base language with gcc 12.2.

¹¹ The entire dolorem-llvm source code can be found in this GitHub repository: <https://github.com/shenniger/dolorem>

41:20 The Dolorem Pattern

Therefore, the `function` macro can not be implemented like in `dolorem-c`, and `dolorem-llvm` does not define one, but rather implements `defun`, and `defmacro` in its base language. There are separate macros to define lambdas.

In `dolorem-c`, the `defun` macro is essentially composed of two different macros, namely the `function` macro which generates C code for the function, and the `compile` macro which hands the generated code to a C compiler.

This shows that, while `dolorem-llvm` and `dolorem-c` both try to keep their base language minimal (principle (2.3.5)), LLVM's more complex API often causes macros to be moved into the base language.

6.3 Language usage: Compiling code

Above, we already showed how to execute Dolorem code directly during compile-time (see 4.8). However, principle (2.4.2) says that it should also be possible to compile Dolorem code.

In order to do that, we need to write LLVM IR code to disk, and then compile it:

1. LLVM only supports to write modules to disk, so by directly calling LLVM's `LLVMModuleCreateWithName`, we create a new LLVM module.
2. For each single function that is to be written to disk, we call `dolorem-llvm`'s `copy-symbol-to-module`. This function looks up a symbol in an array that `dolorem-llvm` copies all function modules to before they are JIT-compiled and copies it to the new module. Note that this works because, like `dolorem-c`, `dolorem-llvm` compiles each function separately and in its own `LLVMModule` (see principle (2.3.4)).
3. Call an LLVM function that exports a module, e.g. `LLVMWriteBitcodeToFile`.
4. Finally, we compile this module to machine code. The easiest way to do this is to invoke `clang` on the command line, but this could also be done using the LLVM library.

Here is a simple example that compiles a “Hello, world!” program to a Linux executable¹²:

```
(defun main () i32 (progn
  (puts "Hello, world!")
  0))
(defmacro compile (progn
  (:= mod (LLVMModuleCreateWithName "test"))
  (copy_symbol_to_module "main" mod)
  (LLVMWriteBitcodeToFile mod "tmp.bc")
  (system "clang tmp.bc -o hello-world";
  (empty-rtv)
  ))
(compile)
```

A program compiled in this way does not depend on the compiler in any way; it is a freestanding executable, quite similar to an executable a C compiler would have created for the same program.

6.4 Precompilation

We can use this mechanism for precompilation.

We compile all functions of a file (including macros) to a shared object. Next time we compile the same file, we still read it entirely and execute all macros, but do not compile any

¹² `(empty-rtv)`, like `dolorem-c`'s `(make-cexp "" "" "" "")`, creates an empty macro return value.

functions within the file if it is unchanged. Instead, we simply load the shared object from last time. This way, we continue to populate all internal data structures like function lists, but avoid calling into LLVM, which is the most time-consuming step.

We would typically use this on macros that define the language. Not having to recompile them every time we compile any `dolorem-llvm` code reduces compilation time.

This mechanism is important because it means that, unlike other staged languages, `dolorem-llvm` does not need to recompile the entire language with every translation unit.

6.5 Optimizations

In section 4.15, we discussed the implementation of code transformations and optimizations in the Dolorem system. We also already explained four basic principles of Dolorem source code transformations.

We will now attempt to use LLVM to optimize `dolorem-llvm` code. To do so, we use the four principles. Our attempt here differs from the attempt in section 4.15 in two main respects: First, since `dolorem-llvm` does not generate code as text, but instead constructs LLVM's data structure which is powerful and easy to modify, we will modify this data structure rather than text. Second, LLVM provides a number of facilities to implement and run optimizations which we can access because we have full access to LLVM (as part of 2.4.1).

That is why, unlike in `dolorem-c`, we will not implement our own layer for optimizing a function, but will instead show how to apply LLVM's `PassManager` to a `dolorem-llvm` function. To do so, we add a macro `optimize` that first lowers its argument, then calls into LLVM to run an optimization¹³:

```
(defmacro optimize (progn
  (:= result (lower (car args)))
  (LLVMBuildRetVoid bldr)
  (LLVMRunPasses mod # the current module
    "instcombine" # the pass name
    (GetThisMachineStdTargetRef) # a convenience function
    defined before
    (LLVMCreatePassBuilderOptions))
  result))
```

Note that we do not directly change the result of `(lower (car args))` before we return it. That is because it merely contains a pointer to an `LLVMValue` of the function return value (and the pointer will not change as part of the optimizations), while the actual sequence of commands is saved in the current `LLVMModule` as a side effect of lowering. This current `LLVMModule` is exposed to macros by the compiler through global variable `mod` and is guaranteed to only contain the currently compiled function (and any nested functions).

The idea of `optimize` is that we can wrap any function body in it to optimize that function, for example:

```
defun do-some-math () void (optimize (progn
  (:= i 10)
  (printf "10 * 16 = %i\n" (* i 16))
));
```

¹³ At the time when `optimize` is called, the final `ret` instruction is likely not yet generated, so we add it to the function before optimization by calling `LLVMBuildRetVoid`.

Since we chose "instcombine" above, LLVM will optimize arithmetic instructions, and thus (similar to the toy optimization we implemented for dolorem-c, but much more powerful) transform our multiplication into a bit shift.

Typically, optimization passes would be added by a flag or some other global compiler setting. Our solution is different: It is only applied to the specific function we want it applied on and also allows us more flexible control in other ways. Finally, our solution allows us to implement our own LLVM optimizer pass and then apply it using the method shown above. We do not show a full example for how, because there is nothing Dolorem-specific about the implementation. It is enough to write the LLVM optimizer function as a normal Dolorem function, then register it in LLVM's `PassManager` in the macro and use it (as shown above). The macro will be able to see the function definition because of principle 2.3.4 and get a pointer to the function, which LLVM can then call.

7 Discussion of the LLVM implementation

We will now evaluate four key factors of our implementation: more complex/powerful language, low code size, minimal/no run-time overhead, and fast compilation times.

7.1 More powerful language

dolorem-llvm is a more powerful and complete language than dolorem-c. It includes its own type system, can interface with C and enables a macro writer to access the full power of LLVM.

To show that we can implement nontrivial example programs in dolorem-llvm, we implemented a graphical Pong. The 192-line-long program uses the game library SDL2 to render graphics and receive inputs.

7.2 Code size

Using our own type system rather than piggybacking on C and emitting the more complex LLVM IR requires slightly more complexity in dolorem-llvm.

Minus header files, the hashmap implementation, the driver and the reader/parser, the entire compiler has about 1980 lines of code. Most of the increase compared to dolorem-c is due to the bigger base language, and also due to the higher complexity of LLVM code.

This shows that, while more complex targets do increase the size of the base language, the basic principle of bootstrapping most of the language still works.

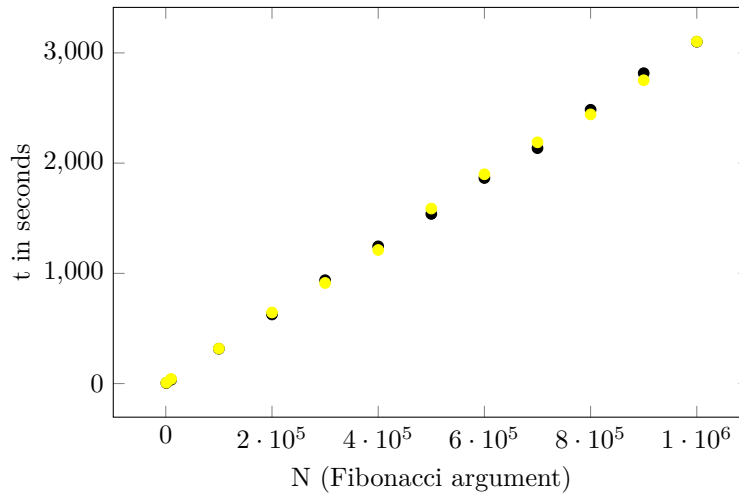
7.3 Run-time overhead

Dolorem's design attempts to impose minimal inherent run-time overhead on programs.

It is challenging to evaluate this goal in general. Primarily, that is because the performance of a Dolorem program depends on the macros used to compile it more than on anything else. In C++, if someone writes a template that is wasteful, that leads to a slower program – just like a Dolorem macro that is badly written or implements a highly ambitious and high-overhead language feature will lead to a slower program.

While it is possible to write programs without complex templates in C++, it is not possible to write non-trivial Dolorem programs without using many macros (that is because we keep the base language minimal, see 2.3.5).

■ **Figure 2** Performance of a Fibonacci function (executed 10,000 times, milliseconds) implemented in C (black) and dolorem-llvm (yellow).



Therefore, in this section, we do not show that all Dolorem environments will be zero-overhead. We consider this to be not only impossible to show given the level afforded to macros, but also to be an undesirable goal: sometimes, using a macro that imposes a little bit of overhead might be a desirable choice, and that is fine, as long as it is a conscious, voluntary decision on the part of the developer.

Instead, we will show that, for our test cases, dolorem-llvm imposes no *inherent* run-time overhead, i.e. that dolorem-llvm allows the creation of a zero-overhead environment using macros and that it is possible to write non-trivial programs in it.

We will show this for two programs written using the default set of macros (i.e. the set of macros we have talked about in this paper). As part of our benchmark, we compile the generated code for a fibonacci function and compare it to an equivalent C implementation, using maximum LLVM optimization for both languages, i.e. `-O3` for C.

We see that the speed is exactly the same (Figure 2).

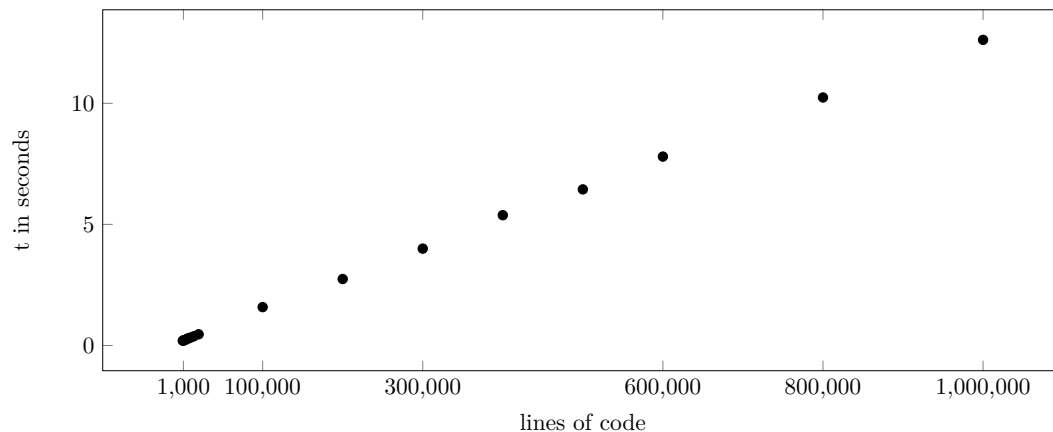
Another test shows the same result: When we compile the graphical Pong we implemented using the mechanism above and measure against an identical program written in C, we see that both are equally fast (in terms of frames per second).

Therefore, we conclude that, while additional tests are necessary to check this promise in the general case, we see that Dolorem exhibits no run-time overhead on the generated code in the cases we tested.

7.4 Compile speed

Many staged languages suffer from compile time problems, because the entire language needs to be recompiled with every translation unit. Dolorem has a unique strength that becomes relevant here: All language macros (like `if`, `add` etc.), after having been compiled, become part of the Dolorem environment, and, like any part of the Dolorem environment, can not only be JIT-compiled, but also exported as machine code (see “Do not prescribe a model of execution”, section 2.4.2). Therefore, unlike in other staged programming languages, it is trivial to precompile any part of the environment (see section 6.4) – including any macros that define the language. Once precompiled, all of those language macros are only a single `dlopen` away.

■ **Figure 3** Performance of dolorem-llvm’s compiler (seconds) for a program of size N lines of code.



That is why we do not include benchmarks on loading the basic environment: basic language macros only need to be compiled once when Dolorem is first installed and then the precompiled versions can be used.

However, we are still interested in the speed of dolorem-llvm’s compiler for new user code.

In order to benchmark this, we need a way of creating large amounts of code that is similar in structure to real-world dolorem-llvm code. We write a macro that duplicates existing code from a number of sources (including the standard library, the Pong implementation and a few other smaller test cases) while changing all names to avoid name clashes.

This way, we simulate how fast dolorem-llvm compiles large, real-world programs of a certain size (Figure 3).

In the case of the Pong example (which contains about 500 lines of Dolorem code in total), the compilation takes about 300 ms. Note that, because we do not use precompilation for this experiment, much of this is LLVM’s start-up time, so, unlike with dolorem-c, total start-up time does not scale proportionally to the amount of code.

We can see from Figure 3 that compilation/start-up times scale linearly with code size. Taking into account that general initialization takes about 200ms, every additional 1000 lines of code take about 12ms. This is competitive with modern C++ compilers.

Precompilation massively helps additionally reduce compile speed. With precompilation enabled, even mid-size examples like Pong start instantly (< 50 ms start-up time) in direct execution mode.

8 Related Work

We discuss various approaches to achieving extensible languages. Dolorem is distinct in that it is a heterogeneous metaprogramming system which gives its macros access to the entire language. Other approaches differ:

Ziggurat [3] is a language tower system to use Scheme-like macros for languages like C. While Ziggurat has a similar aim, namely to add layer-based metaprogramming to low-level languages, Ziggurat’s macros are different: they don’t have access to calling functions, and act more like a preprocessor, limited to generating code. In contrast, Dolorem adheres to the principle (2.3.4) that each macro has access to the full language: a macro can call into functions, for example.

McMicMac [6] is a macro-based system for generative programming in Scheme. It provides a standard way for writing syntax transformations. Dolorem is about expanding the scope of what macros can do, while McMicMac is about putting macros to use.

Converge [12] is a system to embed DSLs in a programming language with a compile-time meta-programming facility. Compared to Dolorem, the setting is homogeneous. The system can guarantee that these embeddings are safe and hygienic (in the sense of LISP macros [5]).

Like Dolorem, MetaOCaml [4] allows a developer to ask the run-time system to compile a piece of code, and link it back to the running program. However, unlike Dolorem, it is a homogeneous system. In practice, MetaOCaml can be seen as a two-stage language, where code is manually marked by quotations to ask the runtime system to compile it. The markings are not essential: inside and outside the markings, code is in the same language. They could even technically be erased yielding the same meaning with a different performance. In contrast, Dolorem is more of a metaprogramming system, where not only what is compiled can be controlled from within the language, but also *how* it is compiled.

Terra [1] and Dolorem share their aim to allow for generative low-level programming using staging. To do so, Terra is staged from within the popular scripting language Lua. Users can generate Terra code from within Lua. Terra and Dolorem also have similar mechanisms to export finished binaries programmatically. However, unlike Terra, Dolorem uses only one language and is mostly bootstrapped from within itself. Most importantly, this means that Dolorem macros have access to the entire language, while Terra and Lua are still separate, although they share the same lexical environment.

Racket [11], like Dolorem, allows macros to change the semantics of the language. Unlike Dolorem macros, Racket macros translate to standard Racket rather than directly generating code.

Metaphor [8] is an object-oriented multi-staging system that targets the .NET Runtime. Metaphor's primary design goal is to allow for type safety. While Metaphor's higher order functions have access to much of the .NET API, they mostly use it for reflection, rather than explicitly generating code.

9 Conclusion

We have described the Dolorem pattern for very flexible metaprogramming and have seen that it exhibits low overhead. Furthermore, we have shown and implemented two languages that make use of the pattern, one that lowers to C and one that lowers to LLVM.

We have discussed the design constraints that the pattern imposes on a language and presented a number of considerations, such as using S-expressions, allowing macros to call into the code generator, and compiling each function separately. We have explained the importance of the `lower` macro and how it compares to traditional `eval`-based approaches.

While `dolorem-c` mostly served as a proof of concept of the pattern, it could already show that our pattern can deliver on its promises of allowing easy language extensions. Its metaprogramming facilities were shown to be powerful enough to implement all arithmetic operators in less than twenty lines of code.

With `dolorem-llvm`, we have shown that the Dolorem pattern can work on a larger scale without losing its core advantages. To show that `dolorem-llvm` is fast and can interface well with existing libraries, we implemented a graphical Pong game.

Next steps

This paper showed a number of macro examples, but most of them were bootstrapping relatively standard language features. It would be interesting to create more complex layers and macros that leverage Dolorem systems to implement more advanced functionality, like instrumentation, data serialization, and borrow checking.

While we have seen in our experiments that Dolorem systems can exhibit low overhead, it would be interesting to conduct more and more methodical experiments on performance, especially those that help better understand how design choices made in macros impact overhead.

It could also be insightful to implement a Dolorem language that targets a more low-level language. For example, there could be a `dolorem-x86` that translates to x86 machine code (or x86 assembly). That would create a new use-case for macros: adding support for new instruction set extensions. It would also invite more experimentation with implementing custom optimization passes as macros.

Furthermore, an interesting topic of future research is to implement cross-compilation within a Dolorem system. `dolorem-c` particularly could be extended by a `cross-compile` form that acts similar to `compile`, except that it compiles for a different architecture. It might also be possible to stage an entirely new language from within a Dolorem language, e.g. to compile OpenGL shaders.

It should also be investigated whether `compile` speeds can be further improved by changing how the C compiler is invoked or by optimizing LLVM's JIT compiler.

Finally, neither `dolorem-c` and `dolorem-llvm` currently have a clearly defined memory model convention. There should be a design contract on where memory is allocated and freed within the compiler (and, by extension, within macros). By creating a design contract, more optimizations and more safety checks would be made possible.

References

- 1 Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A multi-stage language for high-performance computing. *SIGPLAN Not.*, 48(6):105–116, June 2013. doi:10.1145/2499370.2462166.
- 2 Gabriel Dos Reis and Bjarne Stroustrup. General constant expressions for system programming languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10*, pages 2131–2136, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1774088.1774537.
- 3 David Fisher and Olin Shivers. Building language towers with ziggurat. *Journal of Functional Programming*, 18(5-6):707–780, September 2008. doi:10.1017/S0956796808006928.
- 4 Oleg Kiselyov. The design and implementation of ber metaocaml. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 86–102, Cham, 2014. Springer International Publishing.
- 5 Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP '86*, pages 151–161, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/319838.319859.
- 6 Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Generative and Component-Based Software Engineering*, pages 105–120, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

- 7 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.
- 8 Gregory Neverov and Paul Roe. Metaphor: A multi-stage, object-oriented programming language. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering*, pages 168–185, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- 9 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, June 2012. doi:10.1145/2184319.2184345.
- 10 Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1-2):211–242, October 2000. doi:10.1016/S0304-3975(00)00053-0.
- 11 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. *SIGPLAN Not.*, 46(6):132–141, June 2011. doi:10.1145/1993316.1993514.
- 12 Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30(6), October 2008. doi:10.1145/1391956.1391958.

Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types

Sung-Shik Jongmans ✉

Department of Computer Science, Open University, Heerlen, The Netherlands
Centrum Wiskunde & Informatica (CWI), NWO-I, Amsterdam, The Netherlands

Francisco Ferreira ✉

Department of Computer Science, Royal Holloway, University of London, UK

Abstract

Programming distributed systems is difficult. Multiparty session typing (MPST) is a method to automatically prove safety and liveness of protocol implementations relative to protocol specifications.

In this paper, we introduce two new techniques to significantly improve the expressiveness of the MPST method: projection is based on *implicit* local types instead of explicit; type checking is based on the *operational semantics* of implicit local types instead of on the syntax. That is, the reduction relation on implicit local types is used not only “a posteriori” to prove type soundness (as usual), but also “a priori” to define the typing rules – *synthetically*.

Classes of protocols that can now be specified/implemented/verified for the first time using the MPST method include: recursive protocols in which different roles participate in different branches; protocols in which a receiver chooses the sender of the first communication; protocols in which multiple roles synchronously choose both the sender and the receiver of a next communication, implemented as mixed input/output processes. We present the theory of the new techniques, as well as their future potential, and we demonstrate their present capabilities to effectively support regular expressions as global types (not possible before).

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases behavioural types, multiparty session types, choreographies

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.42

Category Pearl/Brave New Idea

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.18>

Funding *Sung-Shik Jongmans*: Netherlands Organisation of Scientific Research: 016.Veni.192.103.

1 Introduction

Programming distributed systems is difficult. One of the challenges is to prove that the implementation of *protocols* (message passing) is safe and live relative to the specification. Safety means that “bad” communications never happen: if a communication happens in the implementation, then it is allowed to happen by the specification. Liveness means that “good” communications eventually happen. *Multiparty session typing* (MPST), proposed by Honda et al. [39, 40], is a method to automatically prove safety and liveness of protocol implementations relative to protocol specifications. Figure 1 visualises the idea:

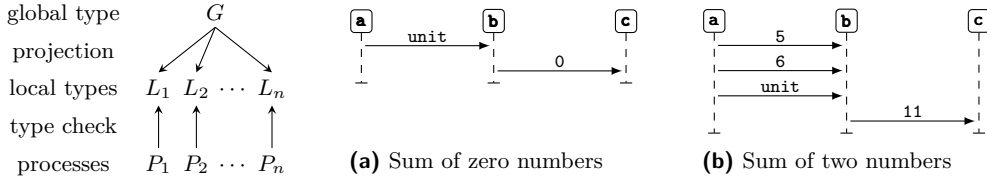
1. First, a *global type* G specifies a protocol among roles/participants r_1, \dots, r_n , while *processes* P_1, \dots, P_n implement it. A global type models the behaviour of all processes together (e.g., “first, a number from Alice to Bob; next, a boolean from Bob to Carol”).
2. Next, *local types* L_1, \dots, L_n are extracted from global type G by *projecting* G onto every role r_i . Each local type models the behaviour of one process alone (e.g., for Bob, “first, he receives a number from Alice; next, he sends a boolean to Carol”).



© Sung-Shik Jongmans and Francisco Ferreira;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 42; pp. 42:1–42:30



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** MPST method.

■ **Figure 2** Two executions of the protocol in Example 1.

3. Last, the processes are verified by *type checking* every process P_i against its local type L_i .

Well-typedness at compile-time implies safety and liveness at run-time.

The following simple example further demonstrates the MPST method.

► **Example 1.** The *Summation* protocol consists of roles *Alice* (**a**), *Bob* (**b**), and *Carol* (**c**). First, zero or more numbers are communicated from Alice to Bob. Next, a token (**unit**) is communicated from Alice to Bob. Last, the sum of the numbers is communicated from Bob to Carol. Figure 2 visualises two executions of this protocol.

The following recursive global type specifies the protocol:

$$G = \mu X. \mathbf{a} \rightarrow \mathbf{b}: \begin{cases} \text{Nat}.X \\ \text{Unit}. \mathbf{b} \rightarrow \mathbf{c}: \text{Nat}. \checkmark \end{cases}$$

Informally, global type $p \rightarrow q: \{t_i.G_i\}_{1 \leq i \leq n}$ specifies the communication of a value of data type t_i from role p to role q , for some $1 \leq i \leq n$; we omit braces when $n = 1$.

The following recursive local types, projected from the global type, specify Alice and Bob:

$$L_{\mathbf{a}} = \mu X. \mathbf{b} \oplus \begin{cases} \text{Nat}.X \\ \text{Unit}. \checkmark \end{cases} \quad L_{\mathbf{b}} = \mu X. \mathbf{a} \& \begin{cases} \text{Nat}.X \\ \text{Unit}. \mathbf{c} \oplus \text{Nat}. \checkmark \end{cases}$$

Informally, local types $q \oplus \{t_i.L_i\}_{1 \leq i \leq n}$ and $p \& \{t_i.L_i\}_{1 \leq i \leq n}$ specify the send and the receive of a value of data type t_i from role p to role q , for some $1 \leq i \leq n$; we omit braces when $n = 1$.

The following processes, well-typed against the local types, implement Alice and Bob:

$$P_{\mathbf{a}} = \underbrace{\bar{\mathbf{b}}\langle 5 \rangle. \bar{\mathbf{b}}\langle 6 \rangle. \bar{\mathbf{b}}\langle \text{unit} \rangle. \mathbf{0}}_{\text{specifically, Figure 1b}} \quad P_{\mathbf{b}} = \text{loop}(\text{sum}: \text{Nat} = 0) \sum \begin{cases} \mathbf{a}(x: \text{Nat}). \text{recur}(\text{sum} + x) \\ \mathbf{a}(_ : \text{Unit}). \bar{\mathbf{c}}\langle \text{sum} \rangle. \mathbf{0} \end{cases}$$

Informally, process $\bar{q}\langle e \rangle.P$ implements the send of the value of expression e to role q , while process $\sum \{p(x_i: t_i). P_i\}_{1 \leq i \leq n}$ implements the receive of a value of data type t_i from role p into variable x_i , for some $1 \leq i \leq n$; we omit \sum and braces when $n = 1$. Well-typedness means that every action implemented in $P_{\mathbf{a}}$ (resp. $P_{\mathbf{b}}$) is also specified in $L_{\mathbf{a}}$ (resp. $L_{\mathbf{b}}$). \lrcorner

Over the past 10–15 years, substantial progress has been made both in MPST theory (e.g., extensions with advanced features, including time [10, 57], security [15–17, 24], and parametrisation [25, 33, 59]) and in MPST practice (e.g., tools for F# [58], F* [71], Go [25], Java [41, 42], OCaml [70], PureScript [46], Rust [48, 49], Scala [26, 61], and TypeScript [56]).

1.1 Open Question: Regular Expressions as Global/Local Types

The expressiveness of the grammar of global/local types determines which protocols can be specified. In turn, this determines which protocols can be implemented in a provably safe and live fashion: the higher the expressiveness, the higher the applicability of the MPST

method to program real(istic) distributed systems. For this reason, substantial research in the community has aimed to increase expressiveness. Doing so is not as simple as just adding new operators to the grammars; to be *effective*, these operators need to be supported by projection and type checking as well, which is actually complicated. As a result, regarding basic features, grammars of global types have effectively evolved as follows:

- In the original paper [39]:

$$G ::= p \rightarrow q : \{t_i . G_i\}_{1 \leq i \leq n} \mid \mu X . G \mid X \mid \checkmark$$

Thus, global types can specify that a sender chooses the data type *but not the receiver*.

- In recent papers [21–23, 54, 65]:

$$G ::= p \rightarrow \{q_i : t_i . G_i\}_{1 \leq i \leq n} \mid \mu X . G \mid X \mid \checkmark$$

Thus, global types can specify that a sender chooses the data type *and also the receiver*.

However, it remains an open question how to effectively generalise these *sub-regular* grammars to *regular* ones (e.g., global types that can specify that a receiver initially chooses the sender). This generalisation would enable the MPST-based verification of significantly more processes.

The notion of using regular expressions as global/local types, or *choreographies*, to specify protocols is intuitive, well-known, and actively studied. Early papers include those by Busi et al. [14], Bravetti–Zavattaro [12, 13], Lanese et al. [50], and Castagna et al. [20]; later papers include those by Guanciale–Tuosto et al. [27, 35, 64], Jongmans et al. [36, 37, 44], and De’Liguoro et al. [29]. Most of these many papers focus on projection, though, while *none* of them focus on type checking: typing rules to verify processes using regular expressions *do not yet exist* in the MPST literature. However, type checking is just as vital as projection in the MPST method (Figure 1). Thus, beyond the non-trivial achievements to *only project* regular expressions, the next elusive milestone is to *also type-check* processes against them.

In summary, the evolution of sub-regular grammars of global/local types has been hard and relatively slow; it also seems to remain relatively far from reaching an effective generalisation to regularity, despite considerable interest in the community. In contrast, for *binary* session typing, the state-of-the-art went beyond regularity already (including *mixed choice* [19]) and has started to explore *context-freeness* [2, 3, 45, 60, 63]. These observations suggest that the open question for multiparty must be significant, too, but apparently very hard to answer. In this paper, we rebuild the foundations of the MPST method using new techniques and answer the open question in the affirmative. For the first time, we effectively generalise the sub-regular grammar of global types to the following “open-ended” regular grammar:

$$G ::= p \rightarrow q : t \mid G_1 + G_2 \mid G_1 \cdot G_2 \mid G^* \mid \checkmark \mid \dots$$

1.2 Contributions of This Paper

In existing papers in the MPST literature, there is a tight correspondence between the structure of global/local types and the structure of processes, instrumental to define projection and type checking. For instance, the global/local types and the processes in Example 1 have essentially the same structure: cosmetics aside, the processes are just syntactic refinements of the global/local types (choices resolved; loops unrolled; values instead of data types).

However, the usage of regular expressions as global/local types breaks the tight correspondence. Generally speaking – deliberately unspecific to regular expressions – the foundational challenge is to define projection and type checking when the grammars are *so far apart* that structurally matching processes to global/local types is prohibitively complicated. *The idea of this paper is to abandon such structural matching and use two new techniques instead:*

- **Local types and projection:** Projection is based on *implicit* local types instead of explicit. To clarify the difference, consider the following projections of global type G in Example 1:

$$L_{\text{old}} = \mu X. \mathbf{a} \& \{ \text{Nat}.X, \text{Unit}.c \oplus \text{Nat}.\checkmark \} \quad L_{\text{new}} = G \upharpoonright \mathbf{b}$$

Explicit local type L_{old} is representative of existing techniques (same as $L_{\mathbf{b}}$ in Example 1): it has the same structure as G . In contrast, implicit local type L_{new} is representative of this paper’s new technique; essentially, it is just a role-indexed global type.

Notably, the concept of *merging*, shown to be problematic for session types [62] (i.e., published results based on merging turned out to be defective), is no longer needed.

- **Type checking:** Type checking is based on the *operational semantics* of implicit local types instead of on the syntax. That is, the reduction relation on implicit local types is used not only “a posteriori” to prove type soundness (as usual), but also “a priori” to define the typing rules. To clarify the difference, consider the following typing rules:

$$\frac{\Xi \vdash e : t_k \quad \Xi \vdash P : L_k}{\Xi \vdash \bar{q}_k \langle e \rangle . P : \sum \{ q_i ! t_i . L_i \}_{1 \leq i \leq n}} \text{[OLD]} \quad \frac{\Xi \vdash e : t \quad \Xi \vdash P : L' \quad L \xrightarrow{q!t} L'}{\Xi \vdash \bar{q} \langle e \rangle . P : L} \text{[NEW]}$$

Rule [OLD] is representative of existing techniques: it states that an output process is well-typed by an explicit local type if it matches the structure. In contrast, rule [NEW] is representative of this paper’s new technique: it states that an output process is well-typed by an implicit local type if it matches the behaviour.¹ As every local type is of the form $G \upharpoonright r$, its reduction relation is derivable from the reduction relation of G . The applicability of rule [NEW] is decidable as the reduction relations constitute finite state machines.

The programmer does not write implicit local types directly, but only global types; implicit local types are automatically extracted as role-indexed global types.

Our aim is to present the theory of the new techniques, as well as their future potential, and to demonstrate their present capabilities:

- X. Protocols that could already be specified/implemented using sub-regular grammars, but not yet verified (i.e., the MPST method is sound but incomplete), *can now be verified*. This includes recursive protocols in which *different roles participate in different branches*.
- Y. Protocols that could already be specified using sub-regular grammars, can now be specified *exponentially more succinct* using regular grammars.
- Z. Protocols that could not yet be specified/implemented/verified using sub-regular grammars, *can now be specified/implemented/verified* using regular grammars. This includes protocols in which *a receiver chooses the sender of the first communication*, and also protocols in which *multiple roles synchronously choose both the sender and the receiver of a next communication* (implemented as mixed input/output processes, similar to `select` for Go channels and POSIX sockets).

We note that the idea of this paper *also* improves the effectiveness of sub-regular grammars (item X). This is because the new techniques are deliberately unspecific to regular expressions, but general: the theory readily supports *any* model of behaviour directly as a global type – be it state-based (e.g., finite automata or labelled transition systems), or event-based (e.g.,

¹ Rule [OLD] is “analytic”: every process/type term that occurs in the premise of a rule *must* also occur as a subterm in the conclusion. In contrast, rule [NEW] is “synthetic” (the dual of “analytic”; e.g., [7, 38]): every process/type term that occurs in the premise of a rule *may* – but does not have to – occur as a subterm in the conclusion. That is, meta-variable L' occurs only in the premise, but not in the conclusion, so it needs to be synthesised to prove well-typedness (by computing the reduction relation).

pomsets or event structures), or logic-based (e.g., CTL or Hennessy–Milner logic) – so long as that model can be interpreted in our general format of operational semantics. Whether or not the usage of such models directly as global types is useful, or preferable over existing algebraic notation, is another research question. But, the future potential seems valuable.

In §2, we further detail the contributions of this paper. In §3, we apply the new techniques to sub-regular grammars. Thus, we introduce the main concepts and complications in a familiar setting. In §4, we apply the new techniques to regular grammars. This section is surprisingly short, which is evidence of the generality of the idea: all complications are addressed in the familiar setting of sub-regular grammars in §3, and those results are almost directly applicable to regular grammars in §4. A separate technical report contains proofs [43].

2 Overview of the Techniques

In this section, using several examples, we further detail the contributions of this paper. The examples follow the three steps of the MPST method (§1), adapted to the new techniques:

- 1a. The programmer writes a global type G and processes P_1, \dots, P_n for roles r_1, \dots, r_n .
- 1b. A tool computes the operational semantics of G and of the implicit local types $G \upharpoonright r_1, \dots, G \upharpoonright r_n$ in the form of a *termination predicate* and a *reduction relation* for every role. Every $G \upharpoonright r_i$ is an implicit local type; it does not compute an explicit one. That is, in this paper, projection is an operator for implicit local types instead of a function on global types.
2. A tool checks if every $G \upharpoonright r_i$ is *well-behaved*. If so, then G is *operationally equivalent* to $G \upharpoonright r_1, \dots, G \upharpoonright r_n$. That is, G mimics $G \upharpoonright r_1, \dots, G \upharpoonright r_n$, and vice versa. Well-behavedness of implicit local types is a new alternative to well-formedness of global types. Importantly, well-behavedness is fully *compositional*: it can be checked separately for every role.
3. A tool checks if every P_i is *well-typed* by $G \upharpoonright r_i$. If so, then $G \upharpoonright r_1, \dots, G \upharpoonright r_n$ is *operationally refined* by P_1, \dots, P_n . That is, $G \upharpoonright r_1, \dots, G \upharpoonright r_n$ mimics P_1, \dots, P_n , but not necessarily vice versa: $G \upharpoonright r_1, \dots, G \upharpoonright r_n$ may specify more behaviour than P_1, \dots, P_n must implement.

2.1 Sub-Regular Grammars

In §3, we apply the new techniques of this paper to the following sub-regular grammars of global types and processes; they are representative of existing ones in the MPST literature:

$$\begin{aligned}
 G &::= p \rightarrow q : \{t_i.G_i\}_{1 \leq i \leq n} \mid \mu X.G \mid X \mid \checkmark & \quad \text{output process} & \quad \text{input process} \\
 P &::= \sum \{O_1, \dots, O_n\} \mid \sum \{I_1, \dots, I_m\} \mid \dots & \quad O ::= \overline{q}(e).P & \quad I ::= p(x:t).P
 \end{aligned}$$

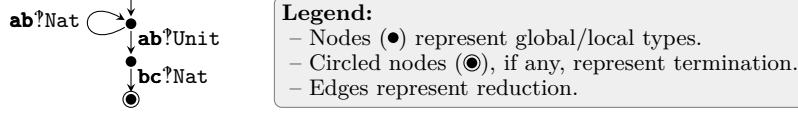
The informal meanings and notational conventions are the same as in Example 1 and further clarified in the examples in this subsection. The examples serve two purposes: to introduce the main concepts, and to demonstrate that the idea of this paper offers distinct expressive power, even in the familiar setting of sub-regular grammars (item **X** in §1.2).

► **Example 2.** We apply steps **1a**, **1b**, **2**, and **3** to the Summation protocol in Example 1:

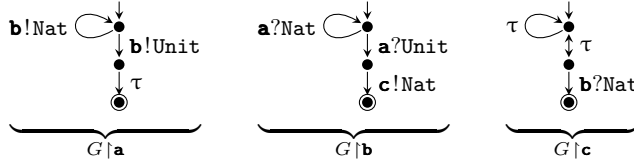
- 1a. The following global type and processes specifies and implement the protocol (same as in Example 1, except the process for Carol, which is new here):

$$\begin{aligned}
 G &= \mu X. \mathbf{a} \rightarrow \mathbf{b} : \begin{cases} \text{Nat}.X \\ \text{Unit}.\mathbf{b} \rightarrow \mathbf{c}:\text{Nat}.\checkmark \end{cases} & \quad \begin{aligned} P_{\mathbf{a}} &= \overline{\mathbf{b}}(5).\overline{\mathbf{b}}(6).\overline{\mathbf{b}}(\text{unit}).\mathbf{0} \\ P_{\mathbf{b}} &= \text{loop}(\text{sum}:\text{Nat}=0) \sum \begin{cases} \mathbf{a}(x:\text{Nat}).\text{recur}(\text{sum}+x) \\ \mathbf{a}(_:\text{Unit}).\overline{\mathbf{c}}(x).\mathbf{0} \end{cases} \\ P_{\mathbf{c}} &= \mathbf{b}(_:\text{Nat}).\mathbf{0} \end{aligned}
 \end{aligned}$$

- 1b. In the style of process algebra, we define a termination predicate and a reduction relation on global/local types to formalise their operational semantics. The following graph visualises the operational semantics of G :



Every reduction is labelled with a *global action* of the form $pq?t$; it models a synchronous communication of a value of data type t from role p to role q . The following graphs visualise the operational semantics of $G\upharpoonright\mathbf{a}$, $G\upharpoonright\mathbf{b}$, and $G\upharpoonright\mathbf{c}$ (the *projections* of G):



Every reduction is labelled with a *local action* of the form $q!t$, $p?t$, or τ ; they model a send, a receive, or “idling” (passage of time in which a role internally waits). The operational semantics of $G\upharpoonright r$ is straightforwardly derived from the operational semantics of G , by replacing every global action $pq?t$ with the corresponding local action: $q!t$ when $p=r\neq q$ (i.e., r is the sender), or $p?t$ when $p\neq r\neq p$ (i.e., r is the receiver), or τ when $p\neq r\neq q$ (i.e., r does not participate in the communication). We note that two τ -reductions are superimposed in the visualisation for Carol; the details do not matter yet (see §3.2).

2. To assure that a global type is operationally equivalent to the family of projections, we define a predicate that analyses the operational semantics of implicit local types, called *well-behavedness*. An implicit local type is well-behaved when:
- **idling is neutral:** the same reductions are possible before and after a τ -reduction;
 - **sending is causal:** a $!$ -reduction is possible initially, or after a $!$ -reduction, or after a $?$ -reduction, or after a τ -reduction when it was possible already before that τ -reduction;
 - **receiving is deterministic:** multiple $?$ -reductions from the same source to different destinations must have different labels.

Thus: *every send must have at least one cause; every receive must have at most one effect*. Our first main result is that if every projection is well-behaved, then the global type is operationally equivalent to the family of projections (Theorem 23). It can be checked that $G\upharpoonright\mathbf{a}$, $G\upharpoonright\mathbf{b}$, and $G\upharpoonright\mathbf{c}$ are well-behaved, so G is operationally equivalent to $\{G\upharpoonright\mathbf{a}, G\upharpoonright\mathbf{b}, G\upharpoonright\mathbf{c}\}$.

3. To assure that a family of projections is operationally refined by a family of processes, we define a typing relation that compares the syntax of processes with the operational semantics of implicit local types. Roughly, $P = \sum\{O_1, \dots, O_n\}$ is well-typed by L when:
- for every O_i , if $O_i = \bar{q}(e)$, then L has a $q!t$ -reduction to L' (modulo τ -reductions);
 - for every $q!t$ -reduction of L (modulo τ -reductions), P has a subprocess O_i .

Furthermore, roughly, $P = \sum\{I_1, \dots, I_m\}$ is well-typed by L when:

- for every I_j , if $I_j = p(x:t)$, then L has a $p?t$ -reduction to L' (modulo τ -reductions);
- for every $p?t$ -reduction of L (modulo τ -reductions), P has a subprocess $I_j = p(x:t)$.

We note that, as usual in the MPST literature, there is asymmetry between well-typedness of selections of output processes and selections of input processes (see §3.6).

Our second main result is that if every process is well-typed by its projection, then the family of projections is operationally refined by the family of processes (Theorem 39). It can be checked that $P_{\mathbf{a}}$ is well-typed by $G \upharpoonright_{\mathbf{a}}$ (traverse the cycle in $G \upharpoonright_{\mathbf{a}}$ twice), $P_{\mathbf{b}}$ is well-typed by $G \upharpoonright_{\mathbf{b}}$, and $P_{\mathbf{c}}$ is well-typed by $G \upharpoonright_{\mathbf{c}}$ (traverse the downwards τ -reduction in $G \upharpoonright_{\mathbf{c}}$; this is sound), so $\{G \upharpoonright_{\mathbf{a}}, G \upharpoonright_{\mathbf{b}}, G \upharpoonright_{\mathbf{c}}\}$ is operationally refined by $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{c}}\}$.

Together, operational equivalence (step 2) and operational refinement (step 3) imply that $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{c}}\}$ is safe and live relative to G (Corollary 41). \square

The Summation protocol is simple. However, it is not yet supported by existing techniques based on global types and projection in the MPST literature. For instance, G in Example 2 is grammatical in the state-of-the-art papers of Majumdar et al. [54] and Van Glabbeek et al. [65], but the projection onto Carol is undefined by Majumdar et al. and ill-formed due to unguarded recursion by Van Glabbeek et al.; as a result, in those papers, G cannot be used to verify processes. The following example further demonstrates the expressive power of our development.

► **Example 3.** The *Recursive Two Buyer* protocol [62] (extension of the *Two Buyer* protocol [39]) consists of roles *Alice* (\mathbf{a}), *Bob* (\mathbf{b}), and *Seller* (\mathbf{s}). Sequentially, it has three subprotocols:

- **Alice and Seller (part 1):** First, the name of an item (**String**) is communicated from Alice to Seller. Next, the price (**Nat**) is communicated from Seller to Alice.
- **Alice and Bob:** When Alice wants to negotiate with Bob to split the price, an offer (**Nat**) is communicated from her to him. Next, an acceptance (**Acc**) is communicated from Bob to Alice and the subprotocol ends, or a rejection (**Rej**) and the subprotocol loops. When Alice wants not to negotiate, a rejection of the sale is communicated from her to him.
- **Alice and Seller (part 2):** When the negotiation between Alice and Bob succeeded (resp. failed), an acceptance (resp. rejection) of the sale is communicated from Alice to Seller.

We apply steps **1a**, **1b**, **2**, and **3** to the Recursive Two Buyer protocol:

1a. The following global type specifies the protocol:

$$G = \mathbf{a} \rightarrow \mathbf{s} : \text{String} . \mathbf{s} \rightarrow \mathbf{a} : \text{Nat} . \mu X . \mathbf{a} \rightarrow \mathbf{b} : \begin{cases} \text{Nat} . \mathbf{b} \rightarrow \mathbf{a} : \{ \text{Acc} . \mathbf{a} \rightarrow \mathbf{s} : \text{Acc} . \checkmark , \text{Rej} . X \} \\ \text{Rej} . \mathbf{a} \rightarrow \mathbf{s} : \text{Rej} . \checkmark \end{cases}$$

The following processes implement Alice, Bob, and Seller:

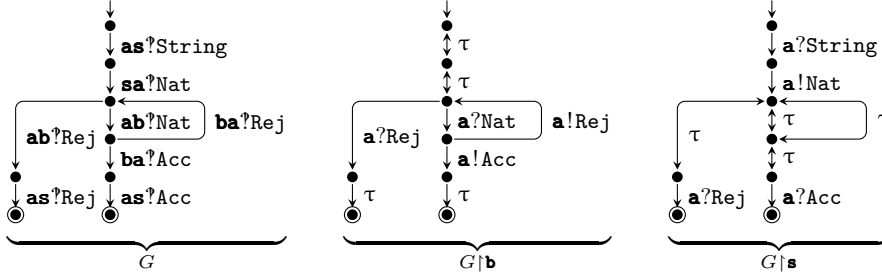
$$P_{\mathbf{a}} = \bar{\mathbf{s}} \langle \text{"foo"} \rangle . \mathbf{s} (x : \text{Nat}) . \bar{\mathbf{b}} \langle x/2 \rangle . \sum \begin{cases} \mathbf{b} (_ : \text{Acc}) . \bar{\mathbf{s}} \langle \text{acc} \rangle . \mathbf{0} \\ \mathbf{b} (_ : \text{Rej}) . \bar{\mathbf{b}} \langle x/3 \rangle . \sum \begin{cases} \mathbf{b} (_ : \text{Acc}) . \bar{\mathbf{s}} \langle \text{acc} \rangle . \mathbf{0} \\ \mathbf{b} (_ : \text{Rej}) . \bar{\mathbf{b}} \langle \text{rej} \rangle . \bar{\mathbf{s}} \langle \text{rej} \rangle . \mathbf{0} \end{cases} \end{cases}$$

$$P_{\mathbf{b}} = \text{loop} \sum \{ \mathbf{a} (y : \text{Nat}) . \text{if } y \leq 10 \text{ } (\bar{\mathbf{a}} \langle \text{acc} \rangle . \mathbf{0}) (\bar{\mathbf{a}} \langle \text{rej} \rangle . \text{recur}) , \mathbf{a} (_ : \text{Rej}) . \mathbf{0} \}$$

$$P_{\mathbf{s}} = \mathbf{a} (z : \text{String}) . \bar{\mathbf{a}} \langle \text{price}(z) \rangle . \sum \{ \mathbf{a} (_ : \text{Acc}) . \mathbf{0} , \mathbf{a} (_ : \text{Rej}) . \mathbf{0} \}$$

$P_{\mathbf{a}}$ implements that Alice offers Bob to contribute half the price; when Bob rejects, Alice offers Bob to contribute a third of the price; when Bob rejects again, Alice rejects the sale. $P_{\mathbf{b}}$ implements that Bob is willing to contribute at most ten units of currency.

1b. The following graphs visualise the operational semantics of G , $G \upharpoonright \mathbf{b}$, and $G \upharpoonright \mathbf{s}$:



2. It can be checked that $G \upharpoonright \mathbf{b}$ and $G \upharpoonright \mathbf{s}$ are well-behaved, in the same way as in Example 2. Furthermore, $G \upharpoonright \mathbf{a}$ is trivially well-behaved, as Alice participates in every communication, so the operational semantics of $G \upharpoonright \mathbf{a}$ has no τ -reductions. Thus, G is operationally equivalent to $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$ (Theorem 23).
3. It can be checked that $P_{\mathbf{a}}$ is well-typed by $G \upharpoonright \mathbf{a}$ (by twice traversing the cycle in $G \upharpoonright \mathbf{a}$), $P_{\mathbf{b}}$ is well-typed by $G \upharpoonright \mathbf{b}$, and $P_{\mathbf{s}}$ is well-typed by $G \upharpoonright \mathbf{s}$. Thus, $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$ is operationally refined by $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{s}}\}$ (Theorem 39).

Together, operational equivalence (step 2) and operational refinement (step 3) imply that $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{s}}\}$ is safe and live relative to G (Corollary 41). \lrcorner

The Recursive Two Buyer protocol was introduced by Scalas–Yoshida to demonstrate the limitations of previous papers based on global types and projection [62]: existing techniques do not support recursive protocols in which *different roles participate in different branches*. The solution proposed by Scalas–Yoshida is to remove global types and projection from the MPST method altogether and, instead, manually write explicit local types for Alice, Bob, and Seller (i.e., they effectively avoid the problem instead of solving it). In contrast, using the new techniques of this paper, we can specify such recursive protocols as global types, *and* automatically extract implicit local types from them, *and* automatically verify processes.

2.2 Regular Grammars

In §4, we apply the new techniques of this paper to the following regular grammars:

$$G ::= p \rightarrow q : t \mid G_1 + G_2 \mid G_1 \cdot G_2 \mid G^* \mid \checkmark \quad P ::= \sum \{O_1, \dots, O_n, I_1, \dots, I_m\} \mid \dots$$

The informal meanings are further clarified in the examples in this subsection. The examples serve two purposes: to evidence generality (i.e., no extra main concepts need to be introduced), and to demonstrate that the idea of this paper offers distinct expressive power. This power arises both in the “soft” sense (i.e., protocols that could already be specified, can now be specified exponentially more succinct; item **Y** in §1.2) and in the “hard” sense (i.e., protocols that could not yet be specified/implemented/verified, can now be; item **Z** in §1.2).

► **Example 4.** The Binomial_k protocol consists of roles *Alice* (**a**) and *Bob* (**b**). A choice between red (**Red**) and blue (**Blu**) is communicated from Alice to Bob, k times, independently. The following global types, which are equivalent, specify the protocol for $k = 3$:

$$G = \mathbf{a} \rightarrow \mathbf{b}: \left\{ \begin{array}{l} \text{Red.} \mathbf{a} \rightarrow \mathbf{b}: \left\{ \begin{array}{l} \text{Red.} \checkmark \\ \text{Blu.} \checkmark \end{array} \right. \\ \text{Blu.} \mathbf{a} \rightarrow \mathbf{b}: \left\{ \begin{array}{l} \text{Red.} \checkmark \\ \text{Blu.} \checkmark \end{array} \right. \\ \dots \text{ (similar to the subtree above)} \end{array} \right. \quad G = (\mathbf{a} \rightarrow \mathbf{b}: \text{Red} + \mathbf{a} \rightarrow \mathbf{b}: \text{Blu}) \cdot \\ (\mathbf{a} \rightarrow \mathbf{b}: \text{Red} + \mathbf{a} \rightarrow \mathbf{b}: \text{Blu}) \cdot \\ (\mathbf{a} \rightarrow \mathbf{b}: \text{Red} + \mathbf{a} \rightarrow \mathbf{b}: \text{Blu})$$

Informally, global types $G_1 + G_2$ and $G_1 \cdot G_2$ specify choice and sequencing. \lrcorner

The Binomial _{k} protocol could already be specified using existing sub-regular grammars of global types in the MPST literature. However, due to the usage of a prefixing operator, the size of G_1 in Example 4 is exponential in k . In contrast, due to the usage of a sequencing operator, the size of G_2 in Example 4 is linear in k . Thus, the Binomial _{k} protocol can now be specified exponentially more succinct. The following example demonstrates that another version of Binomial _{k} , which could not yet be specified/implemented/verified, can now be.

► **Example 5.** The *Role-based Binomial _{k}* protocol consists of roles *Alice* (\mathbf{a}) and *Bob* (\mathbf{b}). A unit is communicated from Alice to Bob, or from Bob to Alice, k times, independently. We apply steps **1a**, **1b**, **2**, and **3** to the Role-based Binomial _{k} protocol:

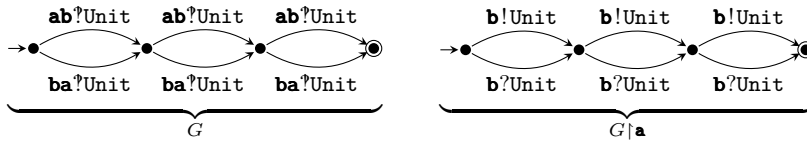
1a. The following global type specifies the protocol for $k = 3$:

$$G = (\mathbf{a} \rightarrow \mathbf{b}: \text{Unit} + \mathbf{b} \rightarrow \mathbf{a}: \text{Unit}) \cdot (\mathbf{a} \rightarrow \mathbf{b}: \text{Unit} + \mathbf{b} \rightarrow \mathbf{a}: \text{Unit}) \cdot (\mathbf{a} \rightarrow \mathbf{b}: \text{Unit} + \mathbf{b} \rightarrow \mathbf{a}: \text{Unit})$$

The following processes implement Alice and Bob:

$$P_{\mathbf{a}} = \sum \left\{ \begin{array}{l} \bar{\mathbf{b}} \langle \text{unit} \rangle. \sum \left\{ \begin{array}{l} \bar{\mathbf{b}} \langle \text{unit} \rangle. 0 \\ \mathbf{b}(_ : \text{Unit}). 0 \end{array} \right. \\ \mathbf{b}(_ : \text{Unit}). \sum \left\{ \begin{array}{l} \bar{\mathbf{b}} \langle \text{unit} \rangle. 0 \\ \mathbf{b}(_ : \text{Unit}). 0 \end{array} \right. \\ \dots \text{ (similar to the subtree above)} \end{array} \right. \quad P_{\mathbf{b}} = \dots \text{ (similar to } P_{\mathbf{a}})$$

1b. The following graphs visualise the operational semantics of G and $G \upharpoonright \mathbf{a}$:



2. It can be checked that $G \upharpoonright \mathbf{a}$ is well-behaved, in the same way as in Example 2. Similarly, $G \upharpoonright \mathbf{b}$ is well-behaved. Thus, G is operationally equivalent to $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}\}$ (Theorem 23). We note that we can use the same definition of well-behavedness as in §2.1, whereas the grammar differs: well-behavedness is independent of structure, so directly re-applicable.

3. Process $P = \sum \{O_1, \dots, O_n, I_1, \dots, I_m\}$ is well-typed by L when:

- $\sum \{O_1, \dots, O_n\}$ is well-typed by L , in the same way as in Example 2;
- $\sum \{I_1, \dots, I_m\}$ is well-typed by L , in the same way as in Example 2.

It can be checked that $P_{\mathbf{a}}$ is well-typed by $G|\mathbf{a}$. In particular, as $P_{\mathbf{a}}$ consists of only three unique subprocesses, no other subprocesses (duplicates) need to be type-checked when memoization is used. The three unique subprocesses are well-typed by the three non-final nodes in the visualisation of $G|\mathbf{a}$. Similarly, $P_{\mathbf{b}}$ is well-typed by $G|\mathbf{b}$. Thus, $\{G|\mathbf{a}, G|\mathbf{b}\}$ is operationally refined by $\{P_{\mathbf{a}}, P_{\mathbf{b}}\}$ (Theorem 39).

Together, operational equivalence (step 2) and operational refinement (step 3) imply that $\{P_{\mathbf{a}}, P_{\mathbf{b}}\}$ is safe and live relative to G (Corollary 41). \dashv

The Role-based Binomial_k protocol could not yet be specified/implemented/verified in previous papers in the MPST literature: existing techniques do not support protocols in which *multiple roles synchronously choose both the sender and the receiver of a next communication*. In contrast, using the new techniques of this paper, we can specify such protocols (e.g., G in Example 5), implement them as *mixed input/output processes* (e.g., $P_{\mathbf{a}}$ and $P_{\mathbf{b}}$ in Example 5), and verify. The following example further demonstrates mixed input/output, and more.

► **Example 6.** The *Acquire–Use–Release* protocol consists of roles *Alice* (\mathbf{a}), *Bob* (\mathbf{b}), and *Server* (\mathbf{s}). Concurrently, it has three subprotocols:

- **Alice and Server (AS):** First, an “acquire” message (**Acq**) is communicated from Alice to Server. Next, a “permission” message (**Perm**) is communicated from Server to Alice. Next, zero or more “usage” messages (**Use**) are communicated from Alice to Server. Last, a “release” message (**Rel**) is communicated from Alice to Server.
 - **Bob and Server (BS):** Similar to **AS**.
 - **Mutual Exclusion (ME):** Between sending “permission” and receiving “release”, Server cannot send another “permission”, thereby constraining the interleaving of **AS** and **BS**.
- We apply steps 1a, 1b, 2, and 3 to the Acquire–Use–Release protocol:

1a. The following global type specifies the protocol:

$$G = + \left\{ \begin{array}{l} \mathbf{a} \rightarrow \mathbf{s} : \text{Acq} \cdot + \left\{ \begin{array}{l} \mathbf{s} \rightarrow \mathbf{a} : \text{Perm} \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Use}^* \cdot + \left\{ \begin{array}{l} \mathbf{a} \rightarrow \mathbf{s} : \text{Rel} \cdot \mathbf{b} \rightarrow \mathbf{s} : \text{Acq} \cdot G'_2 \\ \mathbf{b} \rightarrow \mathbf{s} : \text{Acq} \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Use}^* \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Rel} \cdot G'_2 \end{array} \right. \\ G'_1 \cdot G'_2 \\ G'_2 \cdot G'_1 \end{array} \right. \\ \mathbf{b} \rightarrow \mathbf{s} : \text{Acq} \cdot \dots \text{ (similar to the subtree above)} \end{array} \right.$$

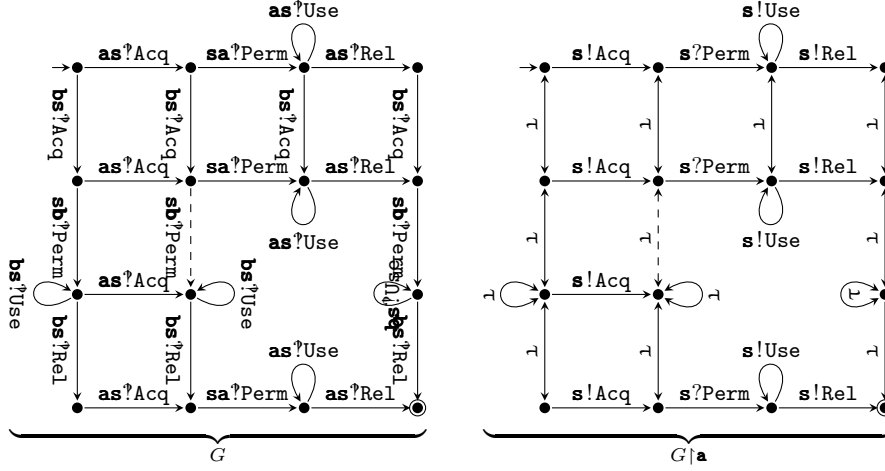
$$G'_1 = \mathbf{s} \rightarrow \mathbf{a} : \text{Perm} \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Use}^* \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Rel} \quad G'_2 = \mathbf{s} \rightarrow \mathbf{b} : \text{Perm} \cdot \mathbf{b} \rightarrow \mathbf{s} : \text{Use}^* \cdot \mathbf{b} \rightarrow \mathbf{s} : \text{Rel}$$

The following processes implement Alice, Bob, and Server:

$$\begin{aligned} P_{\mathbf{a}} &= \bar{\mathbf{s}}\langle \text{acq} \rangle . \mathbf{s}\langle _ : \text{Perm} \rangle . \bar{\mathbf{s}}\langle \text{use} \rangle . \bar{\mathbf{s}}\langle \text{use} \rangle . \bar{\mathbf{s}}\langle \text{use} \rangle . \bar{\mathbf{s}}\langle \text{rel} \rangle . \mathbf{0} \\ P_{\mathbf{b}} &= \bar{\mathbf{s}}\langle \text{acq} \rangle . \mathbf{s}\langle _ : \text{Perm} \rangle . \bar{\mathbf{s}}\langle \text{use} \rangle . \bar{\mathbf{s}}\langle \text{rel} \rangle . \mathbf{0} \\ P_{\mathbf{s}} &= \sum \left\{ \begin{array}{l} \mathbf{a}(\text{acq1} : \text{Acq}) . \sum \left\{ \begin{array}{l} \bar{\mathbf{a}}\langle \text{perm} \rangle . \text{loop} \sum \left\{ \begin{array}{l} \mathbf{a}\langle _ : \text{Rel} \rangle . \mathbf{b}\langle _ : \text{Acq} \rangle . (\dots) \\ \mathbf{b}\langle _ : \text{Acq} \rangle . (\dots) \end{array} \right. \\ \mathbf{b}(\text{acq2} : \text{Acq}) . P_{\mathbf{s}}'' \end{array} \right. \\ \mathbf{b}\langle _ : \text{Acq} \rangle . (\dots) \end{array} \right. \end{array} \right. \\ \text{(version 1)} \quad P_{\mathbf{s}}'' &= \sum \{ \bar{\mathbf{a}}\langle \text{perm} \rangle . (\dots) , \bar{\mathbf{b}}\langle \text{perm} \rangle . (\dots) \} \\ \text{(version 2)} \quad P_{\mathbf{s}}'' &= \text{if } \text{alice_goes_first}(\text{acq1}, \text{acq2}) \text{ } (\bar{\mathbf{a}}\langle \text{perm} \rangle . (\dots)) (\bar{\mathbf{b}}\langle \text{perm} \rangle . (\dots)) \\ \text{(version 3)} \quad P_{\mathbf{s}}'' &= \bar{\mathbf{a}}\langle \text{perm} \rangle . (\dots) \end{aligned}$$

Version 1 of $P_{\mathbf{s}}''$ implements that, after receiving an “acquire” message from both Alice and Bob, Server chooses non-deterministically between sending a “permission” message to Alice or Bob. Versions 2 and 3 of $P_{\mathbf{s}}''$ implement that Server chooses deterministically. We note that the second choice in $P_{\mathbf{s}}$ is between a send and a receive (mixed input/output).

1b. The following graphs visualise the operational semantics of G and $G \upharpoonright \mathbf{a}$:



The dash pattern on the vertical edges is unimportant at this point (see Example 9).

2. It can be checked that $G \upharpoonright \mathbf{a}$ is well-behaved, in the same way as in Example 2. Similarly, $G \upharpoonright \mathbf{b}$ is well-behaved. Furthermore, $G \upharpoonright \mathbf{s}$ is trivially well-behaved, as Server participates in every communication, so the operational semantics of $G \upharpoonright \mathbf{s}$ has no τ -reductions. Thus, G is operationally equivalent to $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$ (Theorem 23).
3. It can be checked that $P_{\mathbf{a}}$ is well-typed by $G \upharpoonright \mathbf{a}$, $P_{\mathbf{b}}$ is well-typed by $G \upharpoonright \mathbf{b}$, and $P_{\mathbf{s}}$ is well-typed by $G \upharpoonright \mathbf{s}$. Thus, $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$ is operationally refined by $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{s}}\}$ (Theorem 39).

Together, operational equivalence (step 2) and operational refinement (step 3) imply that $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{s}}\}$ is safe and live relative to G (Corollary 41). \square

The Role-based Acquire–Use–Release protocol could not yet be specified/implemented/verified in previous papers in the MPST literature: existing techniques do not support protocols in which *a receiver chooses the sender of the first communication*. In contrast, using the new techniques of this paper, we can specify such protocols (e.g., G in Example 6), implement them as processes (e.g., $P_{\mathbf{a}}$, $P_{\mathbf{b}}$, and $P_{\mathbf{s}}$ in Example 5), and verify.

In this paper, projection (including well-behavedness) and type checking are independent of the syntax of global types; they are dependent only on the operational semantics. The formulations and proofs of our main results are similarly independent. As a result of this independence, our regular grammar of global types is actually “open ended”: it can be readily extended with additional global type operators (closed under regularity), intended to serve as higher-level abstractions to make the specification of protocols easier. As a first demonstration of this extensibility, we freely add the following operators:

$$G ::= \cdots \mid G_1 ; G_2 \mid G_1 \parallel G_2 \mid G_1 \bowtie G_2 \mid [G]_{\gamma_2}^{\gamma_1} \mid \cdots$$

► **Example 7.** The following global type specifies that units are communicated first between Alice, Bob1, and Carol1, and second between Alice, Bob2, and Carol2, in that order; the communication from Bob1 to Carol1, and the communication from Bob2 to Carol2, may happen out-of-order, though.

$$G = (\mathbf{a} \rightarrow \mathbf{b1}:\text{Unit} \cdot \mathbf{b1} \rightarrow \mathbf{c1}:\text{Unit}); \\ (\mathbf{a} \rightarrow \mathbf{b2}:\text{Unit} \cdot \mathbf{b2} \rightarrow \mathbf{c2}:\text{Unit})$$

42:12 Sound, Regular Multiparty Sessions via Implicit Local Types

Informally $G_1; G_2$ specifies a “weak” sequence: it is similar to $G_1 \cdot G_2$, except that *independent* communications in G_1 and G_2 (when disjoint roles participate) can happen out-of-order. \lrcorner

► **Example 8.** We re-apply steps **1a**, **1b**, **2**, and **3** to the Acquire–Use–Release protocol:

1a. The following global type, which is equivalent to G in Example 6, specifies the protocol:

$$\begin{aligned}
 G_{AS} &= \mathbf{a} \rightarrow \mathbf{s} : \text{Acq} \cdot \mathbf{s} \rightarrow \mathbf{a} : \text{Perm} \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Use}^* \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Rel} \\
 G_{BS} &= \mathbf{b} \rightarrow \mathbf{s} : \text{Acq} \cdot \mathbf{s} \rightarrow \mathbf{b} : \text{Perm} \cdot \mathbf{b} \rightarrow \mathbf{s} : \text{Use}^* \cdot \mathbf{b} \rightarrow \mathbf{s} : \text{Rel} \\
 G &= (G_{AS} \parallel G_{BS}) \bowtie G_{ME} \\
 G_{ME} &= + \begin{cases} \mathbf{s} \rightarrow \mathbf{a} : \text{Perm} \cdot \mathbf{a} \rightarrow \mathbf{s} : \text{Rel} \cdot \mathbf{s} \rightarrow \mathbf{b} : \text{Perm} \\ \mathbf{s} \rightarrow \mathbf{b} : \text{Perm} \cdot \mathbf{b} \rightarrow \mathbf{s} : \text{Rel} \cdot \mathbf{s} \rightarrow \mathbf{a} : \text{Perm} \end{cases}
 \end{aligned}$$

Informally, global types $G_1 \parallel G_2$ and $G_1 \bowtie G_2$ specify interleaving and join. In general, join demands that every role complies with both of its operands. In this example, join specifies that subprotocol **ME** in Example 6 constrains the interleaving of subprotocols **AS** and **BS**. That is, the three subprotocols are modularly specified as global types G_{AS} , G_{BS} , and G_{ME} , and composed as intended using \parallel and \bowtie ; the result is an exponentially more succinct – and arguably easier to write – specification than in Example 6.

The same processes $P_{\mathbf{a}}$, $P_{\mathbf{b}}$, and $P_{\mathbf{s}}$, with any version of $P_{\mathbf{s}}''$, as in Example 6 are used.

- 1b.** The same graphs as in Example 6 visualise the operational semantics of G and $G \upharpoonright \mathbf{a}$.
2. As in Example 6, G is operationally equivalent to $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$.
 3. As in Example 6, $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$ is operationally refined by $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{s}}\}$. \lrcorner

► **Example 9.** We apply steps **1a**, **1b**, **2**, and **3** to a restricted version of the Acquire–Use–Release protocol in which, after receiving an “acquire” message from both Alice and Bob, Server must send a “permission” message *first* to Alice and *second* to Bob (static order):

1a. The following global type specifies the protocol:

$$G = [(G_{AS} \parallel G_{BS}) \bowtie G_{ME}]_{\mathbf{bs}^? \text{Perm}}^{\mathbf{as}^? \text{Perm}} \quad G_{AS}, G_{BS}, G_{ME} = \dots \text{ (same as in Example 8)}$$

Informally, global type $[G]_{\gamma_2}^{\gamma_1}$ specifies the prioritisation of global action γ_1 (superscript indicates “high” priority) over global action γ_2 (subscript indicates “low” priority) in G . The same processes $P_{\mathbf{a}}$, $P_{\mathbf{b}}$, and $P_{\mathbf{s}}$, with version 3 of $P_{\mathbf{s}}''$, as in Example 6 are used.

- 1b.** The same graphs as in Example 6 visualise the operational semantics of G and $G \upharpoonright \mathbf{a}$, but without the dashed edges.
2. As in Example 6, G is operationally equivalent to $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$.
 3. As in Example 6, $\{G \upharpoonright \mathbf{a}, G \upharpoonright \mathbf{b}, G \upharpoonright \mathbf{s}\}$ is operationally refined by $\{P_{\mathbf{a}}, P_{\mathbf{b}}, P_{\mathbf{s}}\}$. \lrcorner

► **Remark 10.** *Merging* has historically been crucial to support sufficiently expressive kinds of choice in the MPST literature, but it is not needed in this paper. Instead, the issues that merging-based well-formedness of global types address, are covered by well-behavedness of implicit local types. Example 2 and Example 3 already demonstrated this point. To further illustrate it, Table 1 lists global types for examples of Van Glabbeek et al. [65] and Scalas–Yoshida [62]: the examples of van Glabbeek et al. require merging; the examples of Scalas–Yoshida require a more advanced concept (i.e., they use these examples to demonstrate limitations of merging). Every projection of every global type in Table 1 is well-behaved. \lrcorner

■ **Table 1** Example protocols of Van Glabbeek et al. [65] and Scalas–Yoshida [62].

name	global type
Example 9 [vG21]	$G = (\mathbf{b} \rightarrow \mathbf{se}:\text{Talk})^* \cdot \mathbf{b} \rightarrow \mathbf{se}:\text{Buy} \cdot \mathbf{se} \rightarrow \mathbf{sh}:\text{Order}$
Example 13 [vG21]	$G = ((\mathbf{b1} \rightarrow \mathbf{s1}:\text{Wait})^* \cdot \mathbf{b1} \rightarrow \mathbf{s1}:\text{Order}) \parallel ((\mathbf{b2} \rightarrow \mathbf{s2}:\text{Wait})^* \cdot \mathbf{b2} \rightarrow \mathbf{s2}:\text{Order})$
Example 15 [vG21]	$G = (\mathbf{b} \rightarrow \mathbf{s1}:\text{Order1} \cdot \mathbf{b} \rightarrow \mathbf{s2}:\text{Wait})^* \cdot \mathbf{b} \rightarrow \mathbf{s2}:\text{Order2} \cdot \mathbf{b} \rightarrow \mathbf{s1}:\text{Done}$
OAuth2 [SY19]	$G = (\mathbf{s} \rightarrow \mathbf{c}:\text{Login} \cdot \mathbf{c} \rightarrow \mathbf{a}:\text{Passwd} \cdot \mathbf{a} \rightarrow \mathbf{s}:\text{Auth}) + (\mathbf{s} \rightarrow \mathbf{c}:\text{Cancel} \cdot \mathbf{c} \rightarrow \mathbf{a}:\text{Quit})$
Rec. map/reduce [SY19]	$G = G_1 \cdot (\mathbf{r} \rightarrow \mathbf{m}:\text{Continue} \cdot G_1)^* \cdot \mathbf{r} \rightarrow \mathbf{m}:\text{Stop} \cdot \mathbf{m} \rightarrow \mathbf{w1}:\text{Stop}$ $G_1 = \mathbf{m} \rightarrow \mathbf{w1}:\text{Datum} \cdot \mathbf{w1} \rightarrow \mathbf{r}:\text{Result}$
MP workers [SY19]	$G = \mathbf{s} \rightarrow \mathbf{wa1}:\text{Datum} \cdot (G_1 \parallel (\mathbf{s} \rightarrow \mathbf{wa2}:\text{Datum} \cdot G_2))$ $G_1 = (\mathbf{wa1} \rightarrow \mathbf{wb1} \rightarrow \mathbf{wc1}:\text{Datum} \cdot \mathbf{wc1} \rightarrow \mathbf{wa1}:\text{Result})^* \cdot \mathbf{wa1} \rightarrow \mathbf{wb1} \rightarrow \mathbf{wc1}:\text{Stop}$ $G_2 = (\mathbf{wa2} \rightarrow \mathbf{wb2} \rightarrow \mathbf{wc2}:\text{Datum} \cdot \mathbf{wc2} \rightarrow \mathbf{wa2}:\text{Result})^* \cdot \mathbf{wa2} \rightarrow \mathbf{wb2} \rightarrow \mathbf{wc2}:\text{Stop}$

3 Sub-Regular Grammars

In this section, we apply the new techniques for projection and type checking to sub-regular grammars of global types and processes; they are representative of existing ones in the MPST literature. Thus, we introduce the main concepts and complications in a familiar setting.

As this paper is about “processes that communicate” instead of “data that are communicated”, we leave the data language largely unspecified, except for some notation:

- **Syntax:** Let \mathbb{X} denote a set of *variables*, ranged over by x . Let $\mathbb{V} = \{\text{true}, \text{false}, 0, 1, 2, \dots\}$ denote a set of *values*, ranged over by v . Let $\mathbb{E} = \mathbb{X} \cup \mathbb{V} \cup \{\text{!false}, 2+3, \dots\}$ denote a set of *expressions*, ranged over by e . Let $e[v/x]$ denote *substitution* of v for x in e .
- **Static semantics:** Let $\mathbb{T} = \{\text{Bool}, \text{Nat}, \dots\}$ denote a set of *data types*, ranged over by t . Let $(\mathbb{X} \times \mathbb{T})^*$ denote the set of *data typing contexts* (i.e., lists of variable–type pairs), ranged over by Ξ . Let $\Xi \vdash e : t$ denote *well-typedness* of e by t in Ξ .
- **Dynamic semantics:** Let $\text{eval}(e)$ denote *evaluation* of e ; it can be undefined. For instance, $\text{eval}(2+3) = 5$, but $\text{eval}(2+\text{true})$ is undefined. Undefinedness of $\text{eval}(e)$ is a form of “going wrong” [55]; it can give rise to deadlock (Remark 33), prevented by well-typedness (§3.7).

3.1 Global Types – Syntax

Below, we define the grammar of global/local types and abstract global/local actions.

► **Definition 11.** Let \mathbb{R} denote a set of *roles*, ranged over by p, q, r . Let \mathbb{G} and \mathbb{L} denote the sets of *global types* and (*implicit*) *local types*, ranged over by G and L ; they are induced by the following grammar:

$$G ::= p \rightarrow q : \{t_i \cdot G_i\}_{1 \leq i \leq n} \mid \mu X. G \mid X \mid \checkmark \quad L ::= G \upharpoonright r$$

Let $\mathbb{R} \rightarrow \mathbb{L}$ denote the set of *role-indexed families of local types* (partial functions), ranged over by \mathcal{L} . Let $\mathbb{S} = \mathbb{G} \cup \mathbb{L} \cup (\mathbb{R} \rightarrow \mathbb{L})$ denote the set of *specifications*, ranged over by S . ◻

Global type $p \rightarrow q : \{t_i \cdot G_i\}_{1 \leq i \leq n}$ specifies the synchronous *communication* of a value of data type t_i from role p to role q , for some $1 \leq i \leq n$. Global types $\mu X. G$ and X specify a *recursive protocol*. Global type \checkmark specifies the *empty protocol*. Local type $G \upharpoonright r$ specifies the *projection* of G onto r . Thus, projection is a local type operator instead of a function on global types: $G \upharpoonright r$ does not compute an explicit local type; it is an implicit one. The programmer does not write implicit local types directly, but only global types.

► **Definition 12.** Let $\mathbf{\Gamma} = \{(pq!t, pq?t) \mid p \neq q\}$ and $\mathbf{\Lambda} = \bigcup \{(pq!t, pq?t) \mid p \neq q\} \cup \{\tau\}$ denote the sets of (*abstract*) *global actions* and (*abstract*) *local actions*, ranged over by γ and λ . Let $\mathbf{A} = \mathbf{\Gamma} \cup \mathbf{\Lambda}$ denote the set of (*abstract*) *actions*, ranged over by α . ◻

$$\begin{array}{c}
\frac{}{\checkmark \downarrow} [\downarrow\text{G-END}] \quad \frac{1 \leq i \leq n}{p \rightarrow q: \{t_i.G_i\}_{1 \leq i \leq n} \xrightarrow{pq!t_i} G_i} [\rightarrow\text{G-COM}] \quad \frac{G[\mu X.G/X] \xrightarrow{\gamma} G'}{\mu X.G \xrightarrow{\gamma} G'} [\rightarrow\text{G-REC}] \\
\frac{G \downarrow}{G \downarrow r \downarrow} [\downarrow\text{L-AT}] \quad \frac{G \xrightarrow{\gamma} G'}{G \downarrow r \xrightarrow{\gamma!r} G' \downarrow r} [\rightarrow\text{L-AT}] \quad pq?t|r = \begin{cases} pq!t & \text{if: } p = r \neq q \\ pq?t & \text{if: } q \neq r = q \\ \tau & \text{if: } p \neq r \neq q \end{cases} \quad \frac{L \xrightarrow{\tau} L' \not\downarrow}{L' \xrightarrow{\tau} L} [\rightarrow\text{L-REV}] \\
\frac{L_r \downarrow \text{ for every } r}{\{L_r\}_{r \in R} \downarrow} [\downarrow\text{L}] \quad \frac{L_p \xrightarrow{pq!t} L'_p \quad L_q \xrightarrow{pq?t} L'_q}{L_r = L'_r \text{ for every } r \notin \{p, q\}} [\rightarrow\text{L1}] \quad \frac{L_{\bar{r}} \xrightarrow{\tau} L'_{\bar{r}}}{L_r = L'_r \text{ for every } r \notin \{\bar{r}\}} [\rightarrow\text{L2}] \\
\{L_r\}_{r \in R} \xrightarrow{pq?t} \{L'_r\}_{r \in R} \quad \{L_r\}_{r \in R} \xrightarrow{\tau} \{L'_r\}_{r \in R}
\end{array}$$

(a) Termination (b) Reduction. Let $G[\mu X.G/X]$ denote unfolding of X into $\mu X.G$ in G .

■ **Figure 3** Operational semantics of sub-regular global/local types.

Local actions $pq!t$ and $pq?t$ model the *send* and the *receive* of a value of data type t from role p to role q ; we omit p or q when it is clear from the context. Local action τ models internal *idling*. Global action $(pq!t, pq?t)$ models a communication; we often write $pq?t$.

3.2 Global Types – Operational Semantics

Below, we define the termination predicate and reduction relation on global/local types.

► **Definition 13.** Let $G \downarrow$, $L \downarrow$, and $\mathcal{L} \downarrow$ denote termination of G , L , and \mathcal{L} . Formally, \downarrow is the predicate induced by the rules in Figure 3a, while $\not\downarrow$ is its complement (not derivable). ◻

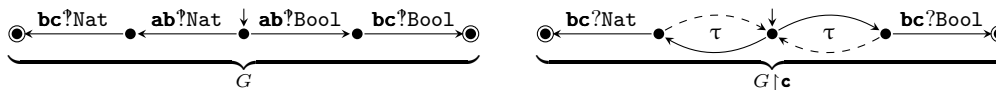
► **Definition 14.** Let $G \xrightarrow{\gamma} G'$, $L \xrightarrow{\lambda} L'$, and $\mathcal{L} \xrightarrow{\lambda_p, \lambda_q} \mathcal{L}'$ denote reduction from G to G' with γ , from L to L' with λ alone, and from \mathcal{L} to \mathcal{L}' with λ_p and λ_q together (synchronously); we omit the label and/or the destination of a reduction if it does not matter. Formally, \rightarrow is the relation induced by the rules in Figure 3b, while $\not\rightarrow$ is its complement (not derivable). ◻

- Rule $[\rightarrow\text{G-COM}]$ states that a communication can reduce with a global action chosen from the alternatives. Following the recent paper of Gheri et al. [34], and for the same reason as them, we omit a reduction rule for out-of-order execution of independent global actions; its interplay with recursion may give rise to infinite reduction relations (e.g., [34, Exmp. 5.1]). We recover out-of-order execution in §4, as already demonstrated in Example 7.
- Rule $[\rightarrow\text{G-REC}]$ states that a recursive protocol can reduce when its body can.
- Rule $[\rightarrow\text{L-AT}]$ states that a projection can reduce when the global type can.
- Rule $[\rightarrow\text{L-REV}]$ states that a τ -reduction into a non-terminated branch can be *reversed*: after “doing nothing” (the τ -reduction from L to L'), a role is always permitted to backtrack by “doing more nothing” (the reverse). This rule ensures that a role r *cannot* commit – unilaterally and irrevocably – to a future communication with another role r' by internally “doing nothing” (i.e., morally, the decision to communicate cannot be made by r alone, but only together with r' , so r should not be able to make a premature commitment and get stuck). Conversely, it *can* commit to local termination by internally “doing nothing” (i.e., morally, the decision to locally terminate can be made by r alone).

► **Example 15.** The following global type specifies that either a number is communicated from Alice to Bob, and from Bob to Carol, or a boolean:

$$G = \mathbf{a} \rightarrow \mathbf{b}: \{ \text{Nat}. \mathbf{b} \rightarrow \mathbf{c}: \text{Nat}. \checkmark, \text{Bool}. \mathbf{b} \rightarrow \mathbf{c}: \text{Bool}. \checkmark \}$$

The following graph visualises the operational semantics of G and $G \downarrow \mathbf{c}$:



Dashed edges represent reductions induced by rule $[\rightarrow\text{L-REV}]$.

Without the τ -reductions of rule $[\rightarrow\text{L-REV}]$, for instance, Carol can commit to the receive of a number by internally “doing nothing” (τ -reduction leftwards). Morally, however, this decision cannot be made by Carol alone, but only together with Bob (depending, in turn, on his previous communication with Alice). *With* the τ -reductions of rule $[\rightarrow\text{L-REV}]$, in contrast, Carol cannot commit: after the τ -reduction leftwards, there is still a sequence of τ -reductions rightwards (which Carol can freely make, because they are internal to her, unobservable to Alice and Bob) to receive a boolean. \lrcorner

- Rules $[\rightarrow\text{L1}]$ and $[\rightarrow\text{L2}]$ state that a family can reduce, when two local types can reduce with a matching send/receive pair (synchronously), or when one can reduce by idling.

The following propositions state basic properties of the operational semantics.

► **Proposition 16** (type-level progress). $G \downarrow$, or $G \rightarrow$ (for every G). \lrcorner

► **Proposition 17** (type-level finiteness). $|\{G^\dagger \mid G \rightarrow \dots \rightarrow G^\dagger\}| \in \mathbb{N}$ (for every G). \lrcorner

Type-level progress and finiteness, which follow straightforwardly from Figure 3, will be used to assure liveness of families of well-typed processes and decidability of type checking.

Recall that S ranges over global types, local types, and families of local types (Definition 11), and α over global actions and local actions (Definition 12):

- Let $S \Rightarrow S^\dagger$ denote τ -reachability from S to S^\dagger : either $S = S^\dagger$, or $S \xrightarrow{\tau} \dots \xrightarrow{\tau} S^\dagger$.
- Let $S \Downarrow$ denote weak termination of S : $S \Rightarrow S^\dagger \downarrow$, for some S^\dagger .
- Let $S \xRightarrow{\alpha} S^\natural$ denote weak reduction from S to S^\natural with α : either $\alpha = \tau$ and $S \Rightarrow S^\natural$, or $S \Rightarrow S^\dagger \xrightarrow{\alpha} S^\ddagger \Rightarrow S^\natural$, for some S^\dagger, S^\ddagger .

As a notational convention, we use “ \prime ” to indicate destinations after 1 reduction, while we use “ \dagger ”, “ \ddagger ”, “ \natural ”, and “ \natural ” to indicate destinations after 0-or-more reductions.

3.3 Main Result 1: Well-Behavedness Implies Operational Equivalence

Intuition. The following global type specifies that a unit is communicated *first* from Alice to Bob, and *second* from Carol to Dave, in-order: $\mathbf{a} \rightarrow \mathbf{b}:\text{Unit}.\mathbf{c} \rightarrow \mathbf{d}:\text{Unit}.\checkmark$ (i.e., the independent actions of Alice–Bob and Carol–Dave cannot be executed out-of-order according to the operational semantics in Figure 3; we recover out-of-order execution in §4). However, this protocol is *unrealisable*: fundamentally, it cannot be implemented as a family of processes without additional covert synchronisation between Bob–Carol. This makes the global type effectively useless. Thus, we need a decision procedure to distinguish “bad” global types from “good” global types, to be able to rule out the bad ones from usage. To achieve this, we define sufficient conditions to ensure that a global type is *operationally equivalent* to the family of projections. That is, operational equivalence formalises protocol realisability.

Instead of defining the conditions on the syntax of global types in terms of *well-formedness* (as usual), we define the conditions on the operational semantics of implicit local types in terms of *well-behavedness*. If the operational semantics of every projection of a global type satisfies every condition, then operational equivalence is guaranteed. Conversely, if the operational semantics of any projection violates any condition, then the global type is ruled out. Well-behavedness is fully *compositional*: it can be checked separately for every role.

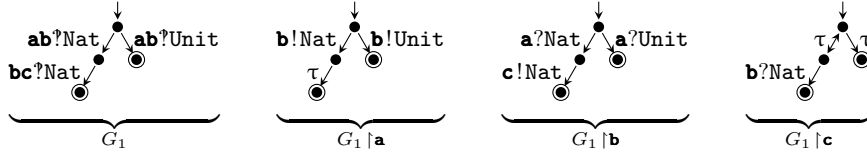
► **Remark 18.** A key advantage of well-behavedness of implicit local types over well-formedness of global types is that it allows us to prove the main results *independently of the set of global type operators*. Thus, the grammar can be extended with new global type operators (such that Propositions 16–17 continue to be valid) without reproving the theorems (§4). ◻

Before defining them formally, we informally introduce the main well-behavedness conditions:

- C1. Idling is neutral:** A local type must always have the same weak termination/reductions before τ -reductions as after them. This means that a role can neither increase nor decrease its behavioural alternatives by idling.
- C2. Sending is causal:** A local type must always have the same strong $!$ -reductions before τ -reductions as after them. This means that if a role can send after idling (later in the future), then it can also send immediately (already in the present). That is, the ability to send cannot arise out of “doing nothing”; there must be an observable cause.
- C3. Receiving is deterministic:** A local type must never have multiple weak $?$ -reductions with the same label but different destinations. This means that if a role receives, then its continuation is uniquely determined. Conditions **C2** and **C3** yield the following duality: *every send must have at least one cause; every receive must have at most one effect.*

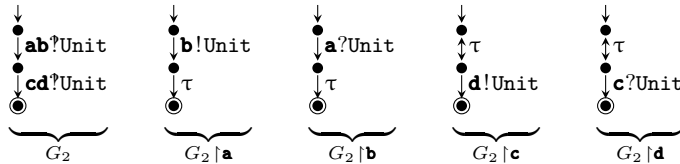
► **Example 19.** We illustrate the conditions with three problematic cases, each of which demonstrates a different reason for operational inequivalence. In each case: first, we define a bad global type that specifies an unrealisable protocol; next, we visualise the operational semantics of it and the projections; next, we argue that they are indeed inequivalent; last, we state the well-behavedness condition that is violated by at least one projection.

- C1.** If $G_1 = \mathbf{a} \rightarrow \mathbf{b} : \{\text{Nat}.\mathbf{b} \rightarrow \mathbf{c} : \text{Nat}.\checkmark, \text{Unit}.\checkmark\}$, then:



The global type cannot be stuck after weak reduction $\xrightarrow{\mathbf{ab}^? \text{Nat}}$: it can always reduce onwards. In contrast, the family of projections can be stuck after weak reduction $\xrightarrow{\mathbf{ab}^? \text{Nat}}$, namely when $G_1 | \mathbf{c}$ weakly reduced rightwards instead of leftwards. In that case, $G_1 | \mathbf{b}$ neither can terminate, nor can reduce onwards (i.e., it needs to synchronise its $!$ -reduction with a $?$ -reduction of $G_1 | \mathbf{c}$, but $G_1 | \mathbf{c}$ has become unable to reciprocate). Thus, G_1 and $\{G_1 | r\}_{r \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}}$ are inequivalent. This is caught by **C1**: $G_1 | \mathbf{c}$ has a weak $?$ -reduction before the rightwards τ -reduction, but not after it, which violates **C1** (i.e., idling is non-neutral), so G_1 is ruled out from usage.

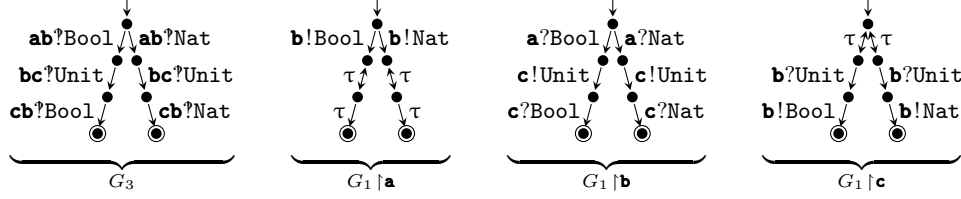
- C2.** If $G_2 = \mathbf{a} \rightarrow \mathbf{b} : \text{Unit}.\mathbf{c} \rightarrow \mathbf{d} : \text{Unit}.\checkmark$, then:



The global type can terminate *only* after weak reductions $\xrightarrow{\mathbf{ab}^? \text{Unit}} \xrightarrow{\mathbf{cd}^? \text{Unit}}$. In contrast, the family of projections can terminate *also* after weak reductions $\xrightarrow{\mathbf{cd}^? \text{Unit}} \xrightarrow{\mathbf{ab}^? \text{Unit}}$, when

$G_2 \upharpoonright \mathbf{c}$ and $G_2 \upharpoonright \mathbf{d}$ begin with $\xrightarrow{\tau} \mathbf{d}!\mathbf{Unit}$ and $\xrightarrow{\tau} \mathbf{c}?\mathbf{Unit}$, and when $G_2 \upharpoonright \mathbf{a}$ and $G_2 \upharpoonright \mathbf{b}$ end with $\xrightarrow{\tau} \mathbf{b}!\mathbf{Unit}$ and $\xrightarrow{\tau} \mathbf{a}?\mathbf{Unit}$. Thus, G_2 and $\{G_2 \upharpoonright r\}_{r \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}}$ are inequivalent. This is caught by **C2**: $G_2 \upharpoonright \mathbf{c}$ does not have a $!$ -reduction before its τ -reduction, but it does have one after it, which violates **C2** (i.e., sending is non-causal), so G_2 is ruled out from usage. We note that if we allowed out-of-order execution of independent global actions, then G_2 would satisfy **C2**. We recover out-of-order execution in §4. The corresponding global type will be $\mathbf{a} \rightarrow \mathbf{b}:\mathbf{Unit}; \mathbf{c} \rightarrow \mathbf{d}:\mathbf{Unit}$, each of whose projections will be well-behaved.

C3. If $G_3 = \mathbf{a} \rightarrow \mathbf{b}:\{\mathbf{Bool}.\mathbf{b} \rightarrow \mathbf{c}:\mathbf{Unit}.\mathbf{c} \rightarrow \mathbf{b}:\mathbf{Bool}.\checkmark, \mathbf{Nat}.\mathbf{b} \rightarrow \mathbf{c}:\mathbf{Unit}.\mathbf{c} \rightarrow \mathbf{b}:\mathbf{Nat}.\checkmark\}$, then:



The global type cannot be stuck after weak reductions $\xrightarrow{\mathbf{ab}?\mathbf{Bool}} \xrightarrow{\mathbf{bc}?\mathbf{Unit}}$: it can always reduce onwards. In contrast, the family of projections can be stuck after weak reductions $\xrightarrow{\mathbf{ab}?\mathbf{Bool}} \xrightarrow{\mathbf{bc}?\mathbf{Unit}}$, namely when $G_1 \upharpoonright \mathbf{c}$ weakly reduced rightwards instead of leftwards. In that case, $G_3 \upharpoonright \mathbf{b}$ neither can terminate, nor can reduce onwards (i.e., it needs to synchronise its $\mathbf{c}?\mathbf{Bool}$ -reduction with a $\mathbf{b}!\mathbf{Bool}$ -reduction of $G_3 \upharpoonright \mathbf{c}$, but $G_3 \upharpoonright \mathbf{c}$ has become unable to reciprocate). Thus, G_3 and $\{G_3 \upharpoonright r\}_{r \in \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}}$ are inequivalent. This is caught by condition **C3**: $G_3 \upharpoonright \mathbf{c}$ has two weak $?$ -reductions with the same label, but to different destinations, which violates **C3** (i.e., receiving is non-deterministic), so G_3 is ruled out from usage. \perp

We relate the well-behavedness conditions on implicit local types in this paper to well-formedness conditions on global types in the MPST literature in the terminology of Castagna et al. [20]. Condition **C1** is usually enforced through projection (i.e., projection determinises explicit local types as they are computed, without using τ -based operators). Condition **C2** is the *sequentiality* principle of Castagna et al.; it is usually enforced by allowing out-of-order execution of independent global actions. Condition **C3** is the *knowledge for choice* principle of Castagna et al. (i.e., a receiver must always be able to uniquely determine which branch the sender was in); it is usually enforced through merging. We note that well-behavedness is relatively permissive regarding branching (some non-directed and non-located choice patterns are allowed), whereas well-formedness is relatively restrictive (all such patterns are forbidden).

Technicalities. First, we define operational equivalence as a relation \approx on specifications (global types, local types, and families of local types). We derive the following requirements from Example 19: **(a)** \approx must be insensitive to idling (i.e., we argued in terms of weak reductions); **(b)** \approx must be sensitive to deadlock (i.e., we distinguished between termination and “being stuck”). Out of many candidates [66, 67], we adopt *weak bisimilarity* (e.g., [68]): it meets both requirements **a** and **b**; additionally, it is sensitive to branching, which is not a requirement, but which makes our proofs easier. Intuitively, two specifications are weak bisimilar when they can mimick each other’s termination/reductions modulo τ -reductions.

► **Definition 20.** Recall that \mathbb{S} denotes the set of all global types, local types, and families of local types, ranged over by S . A weak bisimulation $\heartsuit \subseteq \mathbb{S} \times \mathbb{S}$ is a relation that satisfies the following conditions, for every $(S_1, S_2) \in \heartsuit$, (and for every $S_1^\natural, S_2^\natural, \alpha$):

42:18 Sound, Regular Multiparty Sessions via Implicit Local Types

- If $S_1 \Downarrow$, then $S_2 \Downarrow$. – If $S_1 \xrightarrow{\alpha} S_1^\sharp$, then $S_1^\sharp \heartsuit S_2^\sharp$ and $S_2 \xrightarrow{\alpha} S_2^\sharp$, for some S_2^\sharp .
- If $S_2 \Downarrow$, then $S_1 \Downarrow$. – If $S_2 \xrightarrow{\alpha} S_2^\sharp$, then $S_1^\sharp \heartsuit S_2^\sharp$ and $S_1 \xrightarrow{\alpha} S_1^\sharp$, for some S_1^\sharp .

Let $S_1 \approx S_2$ denote weak bisimilarity. Formally, \approx is the largest weak bisimulation. \lrcorner

Next, we define well-behavedness by formalising the main conditions (plus two more).

► **Definition 21.** Let $\text{wb}(L)$ denote well-behavedness of L . Formally, it is the largest predicate that satisfies the following conditions, for every $L \in \text{wb}$ (and for every $L', L^\dagger, L_1^\dagger, L_2^\dagger, p, q, t$):

- C1.** If $L \Rightarrow L^\dagger$, then $L \approx L^\dagger$. [If L^\dagger is τ -reachable, then L and L^\dagger are weak bisimilar.]
- C2.** If $L \xrightarrow{pq!t} L^\dagger$, then $L \xrightarrow{pq!t} \approx L^\dagger$. [If L has a weak $!$ -reduction to L^\dagger , then it has the same strong $!$ -reduction to a weak bisimilar destination.]
- C3.** If $L \xrightarrow{pq?t} L_1^\dagger$ and $L \xrightarrow{pq?t} L_2^\dagger$, then $L_1^\dagger \approx L_2^\dagger$. [If L has the same weak $?$ -reductions to L_1^\dagger and L_2^\dagger , then L_1^\dagger and L_2^\dagger are weak bisimilar.]
- C4.** If $L \rightarrow$, then $L \not\Downarrow$. [If L can reduce, then it cannot terminate.]
- C5.** If $L \rightarrow L'$, then $\text{wb}(L')$. [Reduction preserves well-behavedness.] \lrcorner

Last, we prove that the conditions of well-behavedness are sufficient to ensure operational equivalence. The idea is to define a *correspondence relation* between global types and families of well-behaved local types. We can then show that correspondence is a weak bisimulation.

► **Definition 22.** Let $G \Leftarrow \{L_r\}_{r \in R}$ denote correspondence of G and $\{L_r\}_{r \in R}$. Formally:

$$\frac{[\text{wb}(G \upharpoonright r) \text{ and } \text{wb}(L_r) \text{ and } G \upharpoonright r \approx L_r]}{G \Leftarrow \{L_r\}_{r \in R}} \quad \lrcorner$$

► **Theorem 23 (equivalence).** If $\text{wb}(G \upharpoonright r)$, for every $r \in R$, then $G \approx \{G \upharpoonright r\}_{r \in R}$. \lrcorner

The proof of this main result is based on two auxiliary lemmas. They state that well-behavedness implies correspondence, and that correspondence implies weak bisimilarity.

► **Lemma 24.** If $\text{wb}(G \upharpoonright r)$, for every $r \in R$, then $G \Leftarrow \{G \upharpoonright r\}_{r \in R}$. \lrcorner

► **Lemma 25.** If $G \Leftarrow \{L_r\}_{r \in R}$, then $G \approx \{L_r\}_{r \in R}$. \lrcorner

The first lemma follows directly from the definition of correspondence and the reflexivity of weak bisimilarity. The proof of the second lemma relies on the definition of well-behavedness.

► **Remark 26.** Theorem 23 depends on premise $\text{wb}(G \upharpoonright r)$. To see that checking this premise is decidable, observe that the reduction relation of G is finite by Proposition 17. As the reduction relation of $G \upharpoonright r$ has exactly the same structure by rules $[\downarrow\text{L-AT}]$ and $[\rightarrow\text{L-AT}]$, and at most linearly many extra τ -transition by rule $[\rightarrow\text{L-REV}]$, it is finite as well. Consequently, checking well-behavedness (including weak bisimilarity [1]) of $G \upharpoonright r$ is trivially decidable. \lrcorner

► **Remark 27.** As an alternative to Theorem 23, of course, it is also possible to check weak bisimilarity between G and $\{G \upharpoonright r\}_{r \in R}$ directly. However, this would require one to compute the reduction relation of $\{G \upharpoonright r\}_{r \in R}$, which is exponentially large in the worst case. In contrast, as well-behavedness is fully compositional, such a computation is avoided. Thus, direct weak bisimilarity is of exponential complexity (in the size of the reduction relations), whereas well-behavedness is of linear complexity and, as a result, better scalable to many roles. \lrcorner

$$\begin{array}{c}
\frac{}{\mathbf{0} \downarrow} [\downarrow\text{P-END}] \\
\frac{P_{\text{eval}(e)} \downarrow}{\text{if } e P_{\text{true}} P_{\text{false}} \downarrow} [\downarrow\text{P-IF}] \\
\frac{P[\mathbf{loop} P/\mathbf{recur}] \downarrow}{\mathbf{loop} P \downarrow} [\downarrow\text{P-LOOP}] \\
\frac{P_r \downarrow \text{ for every } r}{\{P_r\}_{r \in R} \downarrow} [\downarrow\text{P}]
\end{array}
\qquad
\begin{array}{c}
\frac{\overline{pq}\langle e \rangle . P \in \{O_1, \dots, O_n\}}{\sum\{O_1, \dots, O_n\} \xrightarrow{pq! \text{eval}(e)} P} [\rightarrow\text{P-SUM1}] \\
\frac{pq(x:t) . P \in \{I_1, \dots, I_m\} \quad \vdash v : t}{\sum\{I_1, \dots, I_m\} \xrightarrow{pq?v} P[v/x]} [\rightarrow\text{P-SUM2}] \\
\frac{P_{\text{eval}(e)} \xrightarrow{\pi} P' \quad P[\mathbf{loop} P/\mathbf{recur}] \xrightarrow{\pi} P'}{\text{if } e P_{\text{true}} P_{\text{false}} \xrightarrow{\pi} P'} [\rightarrow\text{P-IF}] \quad \frac{P[\mathbf{loop} P/\mathbf{recur}] \xrightarrow{\pi} P'}{\mathbf{loop} P \xrightarrow{\pi} P'} [\rightarrow\text{P-LOOP}] \\
\frac{P_p \xrightarrow{pq!v} P'_p \quad P_q \xrightarrow{pq?v} P'_q \quad P_r = P'_r \text{ for every } r \notin \{p, q\}}{\{P_r\}_{r \in R} \xrightarrow{pq?v} \{P'_r\}_{r \in R}} [\rightarrow\text{P}]
\end{array}$$

(a) Termination. (b) Reduction.

■ **Figure 4** Operational semantics of sub-regular processes. Let $P[v/x]$ denote capture-avoiding substitution of v for x in P . Let $P[\mathbf{loop} P/\mathbf{recur}]$ denote unfolding of **recur** into **loop** P in P .

3.4 Processes – Syntax

Below, we define the grammar of processes and concrete local actions.

► **Definition 28.** Let \mathbb{O} , \mathbb{I} , and \mathbb{P} denote the sets of output processes, input processes, and processes, ranged over by O , I , and P ; they are induced by the following grammar:

$$\begin{array}{l}
P ::= \sum\{O_1, \dots, O_n\} \mid \sum\{I_1, \dots, I_m\} \mid \quad O ::= \overline{pq}\langle e \rangle . P \quad I ::= p(x:t) . P \\
\text{if } e P_1 P_2 \mid \mathbf{loop} P \mid \mathbf{recur} \mid \mathbf{0}
\end{array}$$

Let $\mathbb{R} \rightarrow \mathbb{P}$ denote the set of role-indexed families of processes, ranged over by \mathcal{P} . ┘

Output process $\overline{pq}\langle e \rangle . P$ implements the *send* of the value of expression e from role p to role q ; we omit p when it is clear from the context. Input process $pq(x:t) . P$ implements the *receive* of a value of data type t into variable x from role p to role q ; we omit q when it is clear from the context; we omit “:t” when the data type does not matter. Processes $\sum\{O_1, \dots, O_n\}$ and $\sum\{I_1, \dots, I_m\}$ implement non-deterministic *selections* of n output processes (sends) and m input processes (receives); we omit “ \sum ” and braces when $n = m = 1$. Process **if** $e P_1 P_2$ implements a *conditional choice*. Processes **loop** P and **recur** implement a *loop*. Process **0** implements the *empty process*. We note that data parameters can be added to loops in the standard way (e.g., [62]). Process creation and session creation are orthogonal to the contributions of this paper and thus we omit them.

► **Remark 29.** We stipulate that every process is *guarded* (i.e., **recur** occurs only inside \sum -processes) and *closed* (i.e., **recur** occurs only inside **loop**-processes), while every family $\{P_r\}_{r \in R}$ is *well-formed* (i.e., for every $r \in R$, every output process that occurs in P_r is of the form $\overline{rq}\langle e \rangle . P'$, while every input process is of the form $pr(x:t) . P'$). ┘

► **Definition 30.** Let $\mathbb{II} = \bigcup\{pq!v, pq?v \mid p \neq q\}$ denote the set of (concrete) local actions, ranged over by π . ┘

3.5 Processes – Operational Semantics

Below, we define the termination predicate and reduction relation on processes.

► **Definition 31.** Let $P \downarrow$ and $\mathcal{P} \downarrow$ denote termination of P and \mathcal{P} . Formally, \downarrow is the predicate induced by the rules in Figure 4a. ┘

$$\begin{array}{c}
\frac{\Xi \vdash e : t \quad \Xi, \Upsilon \vdash P : L^\sharp \quad L \xrightarrow{pq!t} L^\sharp}{\Xi, \Upsilon \vdash \overline{pq}(e).P : L} [\vdash\text{-OUT}] \quad \frac{\Xi, x : t, \Upsilon \vdash P : L^\sharp \quad L \xrightarrow{pq?t} L^\sharp}{\Xi, \Upsilon \vdash pq(x:t).P : L} [\vdash\text{-IN}] \\
\frac{\Xi, \Upsilon \vdash O_i : L \text{ for every } 1 \leq i \leq n \quad [O_i = \overline{p}\langle \cdot \rangle. _ \text{ for some } 1 \leq i \leq n] \text{ for every } L \xrightarrow{pq!t}}{\Xi, \Upsilon \vdash \sum\{O_1, \dots, O_n\} : L} [\vdash\text{-SUM1}] \\
\frac{\Xi, \Upsilon \vdash I_j : L \text{ for every } 1 \leq j \leq m \quad [I_j = pq(_ : t). _ \text{ for some } 1 \leq j \leq m] \text{ for every } L \xrightarrow{pq?t}}{\Xi, \Upsilon \vdash \sum\{I_1, \dots, I_m\} : L} [\vdash\text{-SUM2}] \\
\frac{L \Downarrow}{\Xi, \Upsilon \vdash \mathbf{0} : L} [\vdash\text{-END}] \quad \frac{\Xi, \Upsilon, \mathbf{recur} : L \vdash P : L}{\Xi, \Upsilon \vdash \mathbf{loop} P : L} [\vdash\text{-LOOP}] \quad \frac{L_1 \approx L_2}{\Xi, \Upsilon, \mathbf{recur} : L_1 \vdash \mathbf{recur} : L_2} [\vdash\text{-RECUR}] \\
\frac{\Xi \vdash e : \mathbf{Bool} \quad \Xi, \Upsilon \vdash P_1 : L \quad \Xi, \Upsilon \vdash P_2 : L}{\Xi, \Upsilon \vdash \mathbf{if} e P_1 P_2 : L} [\vdash\text{-IF}] \quad \frac{[\vdash P_r : L_r \text{ and } \mathbf{wb}(L_r)] \text{ for every } r}{\vdash \{P_r\}_{r \in R} : \{L_r\}_{r \in R}} [\vdash]
\end{array}$$

■ **Figure 5** Well-typedness (“ $_$ ” is a meta-variable to indicate that the object does not matter).

► **Definition 32.** Let $P \xrightarrow{\pi} P'$ and $\mathcal{P} \xrightarrow{\pi_p, \pi_q} \mathcal{P}'$ denote reduction from P to P' with π alone, and from \mathcal{P} to \mathcal{P}' with π_p and π_q together (synchronously). Formally, \rightarrow is the relation induced by the rules in Figure 4b. \lrcorner

Rule $[\rightarrow\text{-SUM1}]$ (resp. $[\rightarrow\text{-SUM2}]$) states that a selection can reduce with a send (resp. receive) when there is a corresponding output process (resp. input process) among the alternatives and $\mathbf{eval}(e)$ is defined (resp. v is well-typed by t and bound to x). Rule $[\rightarrow\text{-IF}]$ states that a conditional choice can reduce when $\mathbf{eval}(e) \in \{\mathbf{true}, \mathbf{false}\}$ and the corresponding branch can reduce. Rule $[\rightarrow\text{-LOOP}]$ states that a recursive loop can reduce when its body can. Rule $[\rightarrow\text{-P}]$ states that a family can reduce when two processes can reduce with a matching send/receive pair (synchronously).

► **Remark 33.** Figure 4 contains no rules for communication errors: “going wrong” manifests as *deadlock*. There are three situations in which this can happen for a process P or family \mathcal{P} :

- If $P = \mathbf{if} e P_1 P_2$, but $\mathbf{eval}(e) \notin \{\mathbf{true}, \mathbf{false}\}$, then rules $[\downarrow\text{-IF}]/[\rightarrow\text{-IF}]$ are inapplicable.
- If $P = \sum\{\overline{p_1 q_1}(e_1).P_1, \dots, \overline{p_n q_n}(e_n).P_n\}$ and $n > 1$, but $\mathbf{eval}(e_i)$ is undefined for every $1 \leq i \leq n$, then rule $[\rightarrow\text{-SUM1}]$ is inapplicable.
- If not all processes in \mathcal{P} can terminate, while no two processes in \mathcal{P} can reduce with a matching send and receive, then rules $[\downarrow\text{-P}]/[\rightarrow\text{-P}]$ are inapplicable.

In each situation, P or \mathcal{P} cannot terminate/reduce. Well-typedness will prevent this. \lrcorner

3.6 Main Result 2: Well-Typedness Implies Operational Refinement

Now comes the pivotal concept among our contributions: the typing rules are based on the operational semantics of implicit local types instead of on their syntax. That is, the termination predicate and the reduction relation on local types are used not only “a posteriori” to prove type soundness (as usual), but also “a priori” to define the typing rules. This allows us to break the historically tight correspondence between the structure of global/local types and the structure of processes (§1.2).

► **Definition 34.** Recall that $(\mathbb{X} \times \mathbb{T})^*$ denotes the set of data typing contexts, ranged over by Ξ . Let $(\mathbf{recur} \times \mathbb{L})^*$ denote the set of process typing contexts, ranged over by Υ . Let $\Xi, \Upsilon \vdash O : L$, $\Xi, \Upsilon \vdash I : L$, $\Xi, \Upsilon \vdash P : L$ and $\vdash \mathcal{P} : \mathcal{L}$ denote well-typedness of O , I , P by L in Ξ, Υ , and of \mathcal{P} by \mathcal{L} . Formally, \vdash is the relation induced by the rules in Figure 5. \lrcorner

Rule $[\vdash\text{-END}]$ states that the empty process is well-typed when the local type can weakly terminate. Rule $[\vdash\text{-IF}]$ states that a conditional choice is well-typed when the condition and the branches are well-typed. Rules $[\vdash\text{-LOOP}]/[\vdash\text{-RECUR}]$ state that a loop is well-typed when the body and the recursive calls are well-typed. Rule $[\vdash]$ states that a family of processes is well-typed when every process is well-typed by a well-behaved local type.

Rule $[\vdash\text{-OUT}]$ states that an output process is well-typed when the local type has an *analogous* weak $!$ -reduction such that the expression and the continuation are well-typed; “analogous” means “same sender, same receiver, same data type”. Rule $[\vdash\text{-IN}]$ states that an input process is well-typed when the local type has an analogous weak $?$ -transition, and the continuation is well-typed. Rule $[\vdash\text{-SUM1}]$ states that a selection of output processes is well-typed when every subprocess is well-typed, and there is a *possibly non-analogous* subprocess for every weak $!$ -reduction of the local type. Rule $[\vdash\text{-SUM2}]$ states that a selection of input processes is well-typed when every subprocess is well-typed, and there is an *analogous* subprocess for every weak $?$ -reduction of the local type.

► **Remark 35.** As usual in the MPST literature, there is asymmetry between well-typedness of selections of output processes and selections of input processes: if the local type specifies ≥ 1 sends, then the process *may* implement *one* of them (i.e., the programmer statically chooses what/whereto the process sends); if it specifies ≥ 1 receives, then it *must* implement *all* of them (i.e., the environment dynamically chooses what/wherefrom the process receives). ◻

► **Example 36.** Let $G = \mathbf{a} \rightarrow \mathbf{b} : \{\mathbf{Nat}.\mathbf{b} \rightarrow \mathbf{c} : \mathbf{Nat}.\checkmark, \mathbf{Bool}.\mathbf{b} \rightarrow \mathbf{c} : \mathbf{Bool}.\checkmark\}$; the operational semantics of this global type was previously visualised in Example 15.

- **Alice:** Process $\bar{\mathbf{b}}\langle 5 \rangle.\mathbf{0}$, process $\bar{\mathbf{b}}\langle \mathbf{true} \rangle.\mathbf{0}$, and process $\mathbf{if\ cond}() (\bar{\mathbf{b}}\langle 5 \rangle.\mathbf{0}) (\bar{\mathbf{b}}\langle \mathbf{true} \rangle.\mathbf{0})$ are all well-typed by $G \upharpoonright \mathbf{a}$, because rule $[\vdash\text{-SUM1}]$ requires only one send specified to be implemented. Process $\bar{\mathbf{b}}\langle \mathbf{f}\mathbf{o}\mathbf{o} \rangle.\mathbf{0}$ is ill-typed, because rule $[\vdash\text{-SUM1}]$ requires every send implemented to be specified. Process $\mathbf{loop} (\bar{\mathbf{b}}\langle 5 \rangle.\mathbf{recur})$ is ill-typed by $G \upharpoonright \mathbf{a}$ as well, because rule $[\vdash\text{-LOOP}]$ adds $\mathbf{recur} : G \upharpoonright \mathbf{a}$ to the process typing context at the root of the derivation tree, but rule $[\vdash\text{-RECUR}]$ requires $\mathbf{recur} : \checkmark$ at the leaf, and $G \upharpoonright \mathbf{a} \not\approx \checkmark$.
- **Carol:** Process $\mathbf{b}(x:\mathbf{Nat}).\mathbf{0}$ and process $\mathbf{if\ cond}() (\mathbf{b}(x:\mathbf{Nat}).\mathbf{0}) (\mathbf{b}(x:\mathbf{Bool}).\mathbf{0})$ are both ill-typed by $G \upharpoonright \mathbf{c}$, because rule $[\vdash\text{-SUM2}]$ requires every receive specified to be implemented. Process $\sum \{\mathbf{b}(x:\mathbf{Nat}).\mathbf{0}, \mathbf{b}(x:\mathbf{Bool}).\mathbf{0}\}$ is well-typed by $G \upharpoonright \mathbf{c}$. ◻

The asymmetry between the right-sided premises of rules $[\vdash\text{-SUM1}]/[\vdash\text{-SUM2}]$ ensure that if a family of well-typed local types can reduce, then the family of well-typed processes can reduce, too, *but possibly with a non-analogous communication*. This is *progress*:

► **Lemma 37.** *If $G \rightleftharpoons \mathcal{L}$ and $\vdash \mathcal{P} : \mathcal{L}$, then $\mathcal{P} \downarrow$ or $\mathcal{P} \rightarrow$.* ◻

Complementary, the symmetry between the left-sided premises of rules $[\vdash\text{-SUM1}]/[\vdash\text{-SUM2}]$ (i.e., every subprocess of every selection needs to be well-typed) ensure that if a family of well-typed processes can reduce, then the family of well-behaved local types can reduce, too, *and necessarily with an analogous communication*. This is *preservation*:

► **Lemma 38.** *If $G \rightleftharpoons \mathcal{L}$ and $\vdash \mathcal{P} : \mathcal{L}$, then (for every \mathcal{P}', p, q, v):*

- *If $\mathcal{P} \downarrow$, then $G \downarrow$ and $\mathcal{L} \downarrow$.*
- *If $\mathcal{P} \xrightarrow{pq^?v} \mathcal{P}'$, then $G' \rightleftharpoons \mathcal{L}'$ and $\vdash \mathcal{P}' : \mathcal{L}'$ and $\vdash v : t$ and $G \xrightarrow{pq^?t} G'$ and $\mathcal{L} \xrightarrow{pq^?t} \mathcal{L}'$, for some t, G', \mathcal{L}' .* ◻

Progress and preservation entail *operational refinement*: every trace of the family of processes (with concrete actions) is also a trace of the family of projections (with analogous abstract actions); moreover, if the family of processes can terminate or deadlock, then also the family of projections can. We formalise this concept directly in the following theorem.

- **Theorem 39** (refinement). *If $\vdash \mathcal{P} : \{G \upharpoonright r\}_{r \in R}$, and $\mathcal{P} \xrightarrow{p_1 q_1 ? v_1} \dots \xrightarrow{p_n q_n ? v_n} \mathcal{P}^\dagger$, then:*
- $\{G \upharpoonright r\}_{r \in R} \xrightarrow{p_1 q_1 ? t_1} \dots \xrightarrow{p_n q_n ? t_n}$, and $\vdash v_1 : t_1$, and \dots , and $\vdash v_n : t_n$, for some t_1, \dots, t_n .
 - If $\mathcal{P}^\dagger \downarrow$, then $\mathcal{L}^\dagger \downarrow$.
 - If $\mathcal{P}^\dagger \not\downarrow$ and $\mathcal{P}^\dagger \not\rightarrow$, then $\mathcal{L}^\dagger \not\downarrow$ and $\mathcal{L}^\dagger \not\rightarrow$. ┘

► **Remark 40.** Theorem 39 depends on premise $\vdash \mathcal{P} : \{G \upharpoonright r\}_{r \in R}$. To see that checking this premise is decidable, observe that we need to check two sets of properties by rule $[\vdash]$: (1) well-typedness of the processes in \mathcal{P} by the local types in $\{G \upharpoonright r\}_{r \in R}$; (2) well-behavedness of the local types in $\{G \upharpoonright r\}_{r \in R}$. Regarding the first set, the typing rules for processes are defined inductively on the structure of processes. Consequently, the number of applications is finite. Furthermore, checking the premise of rule $[\vdash\text{-IN}]/[\vdash\text{-OUT}]/[\vdash\text{-SUM1}]/[\vdash\text{-SUM2}]$ is decidable because the reduction relation of every local type in $\{G \upharpoonright r\}_{r \in R}$ is finite (Remark 26). Thus, checking the first set of properties is decidable. Regarding the second set, see Remark 26. ┘

3.7 Safety and Liveness

We proved operational equivalence for projection (Theorem 23) and operational refinement for type checking (Theorem 39). Together, these main results entail type soundness.

- **Corollary 41** (type soundness). *If $\vdash \mathcal{P} : \{G \upharpoonright r\}_{r \in R}$ and $\mathcal{P} \xrightarrow{p_1 q_1 ? v_1} \dots \xrightarrow{p_n q_n ? v_n} \mathcal{P}^\dagger$, then:*
- **Safety:** $G \xrightarrow{p_1 q_1 ? t_1} \dots \xrightarrow{p_n q_n ? t_n}$, and $\vdash v_1 : t_1$, and \dots , and $\vdash v_n : t_n$, for some t_1, \dots, t_n .
 - **Liveness:** $\mathcal{P}^\dagger \downarrow$, or $\mathcal{P}^\dagger \rightarrow$. ┘

All communication errors that can give rise to deadlock (Remark 33) are ruled out when $\vdash \mathcal{P} : \{G \upharpoonright r\}_{r \in R}$ holds. Moreover, checking if $\vdash \mathcal{P} : \{G \upharpoonright r\}_{r \in R}$ holds, is decidable (Remark 40).

4 Regular Grammars

In this section, we apply the new techniques for projection and type checking to regular grammars of global types and processes for the first time. To achieve this, we need to revise and extend the definitions of the grammars in §3, termination/reduction rules, and typing rules. In contrast, the definitions of implicit local types, projection, and well-behavedness – as well as the main result of operational equivalence (Theorem 23) – can stay exactly the same as in §3: they were all formulated in general terms of termination and reduction, but not in specific terms of the rules that define them. Thus, only the following changes are needed:

- **Definition 42** (revision of Definition 11). $G ::= p \rightarrow q : t \mid G_1 + G_2 \mid G_1 \cdot G_2 \mid G^* \mid \checkmark$ ┘

Global type $p \rightarrow q : t$ specifies a synchronous communication. Global types $G_1 + G_2$ and $G_1 \cdot G_2$ specify the *choice* between, and the *sequence* of, G_1 and G_2 . Global type G^* specifies the *finite repetition* of G . Global type \checkmark specifies the *empty protocol*.

- **Definition 43** (revision of Definition 13 and Definition 14). *See Figure 6.* ┘

- **Definition 44** (extension of Definition 28). $P ::= \dots \mid \sum \{O_1, \dots, O_n, I_1, \dots, I_m\}$ ┘

Process $\sum \{O_1, \dots, O_n, I_1, \dots, I_m\}$ implements a non-deterministic *selection* of n output processes and m input processes, simultaneously (i.e., it is a *mixed input/output process*).

- **Definition 45** (extension of Definition 32). *See Figure 7.* ┘

$$\begin{array}{c}
\frac{}{\checkmark \downarrow} \quad \frac{G_i \downarrow}{G_1 + G_2 \downarrow} \\
\frac{G_1 \downarrow \quad G_2 \downarrow}{G_1 \cdot G_2 \downarrow} \quad \frac{}{G^* \downarrow}
\end{array}
\qquad
\begin{array}{c}
\frac{}{p \rightarrow q : t \xrightarrow{pq^*t} \checkmark} \quad \frac{G_i \xrightarrow{\gamma} G'}{G_1 + G_2 \xrightarrow{\gamma} G'} \\
\frac{G_1 \xrightarrow{\gamma} G'_1}{G_1 \cdot G_2 \xrightarrow{\gamma} G'_1 \cdot G_2} \quad \frac{G_1 \downarrow \quad G_2 \xrightarrow{\gamma} G'}{G_1 \cdot G_2 \xrightarrow{\gamma} G'} \quad \frac{G \xrightarrow{\gamma} G'}{G^* \xrightarrow{\gamma} G' \cdot G^*}
\end{array}$$

(a) Termination (b) Reduction

■ **Figure 6** Operational semantics of regular global types (standard for regular expressions; e.g., [4]).

$$\cdots \text{ (same as in Figure 4b)} \quad \frac{\sum\{O_1, \dots, O_n\} \xrightarrow{\pi} P'}{\sum\{O_1, \dots, O_n, I_1, \dots, I_m\} \xrightarrow{\pi} P'} \quad \frac{\sum\{I_1, \dots, I_m\} \xrightarrow{\pi} P'}{\sum\{O_1, \dots, O_n, I_1, \dots, I_m\} \xrightarrow{\pi} P'}$$

■ **Figure 7** Operational semantics of regular processes – Reduction rules.

► **Definition 46** (extension of Definition 34). *See Figure 8.* ┘

While Theorem 23 of operational equivalence is directly applicable to the regular grammar of global types in this section, Theorem 39 of operational refinement requires minor effort: we need to prove a new, yet simple, inductive case for the new typing rule in Figure 8. Then:

► **Corollary 47.** *Corollary 41 is applicable to the revisions and extensions in this section.* ┘

Extending the regular grammar of processes with new process operators (including typing rules) requires one to prove additional inductive cases. In contrast, extending the regular grammar of global types with new global type operators (such that Propositions 16–17 continue to be valid) is completely *free*. That is, in this paper, projection (including well-behavedness) and type checking are independent of the syntax of global types; they are dependent only on the operational semantics. The formulations and proofs of our main results are similarly independent. As a result of this independence, our regular grammar of global types is actually “open ended”. As a first demonstration of this extensibility, we freely add a few global type operators; they are intended to serve as higher-level abstractions to make the specification of protocols easier. See also Example 8 and Example 9 in §2.2.

► **Definition 48** (extension of Definition 42). $G ::= \cdots \mid G_1;G_2 \mid G_1 \parallel G_2 \mid G_1 \bowtie G_2 \mid [G]_{\gamma_2}^{\gamma_1}$ ┘

- Global type $G_1;G_2$ specifies the *weak sequence* of G_1 and G_2 . It is similar to $G_1 \cdot G_2$, except that independent communications in G_1 and G_2 can happen out-of-order. Communications are independent when they have disjoint sets of participating roles. By using G^* instead of $\mu X.G$ for loops (wlog for regularity), weak sequencing yields finite reduction relations.
- Global type $G_1 \parallel G_2$ specifies the *interleaving* of G_1 and G_2 . We note that interleaving was already present in the original paper on MPST [39], as well as in later papers (e.g., [20, 30, 31, 51]). However, in these papers, G_1 and G_2 need to have disjoint roles or disjoint channels, whereas in this paper, G_1 and G_2 need to have disjoint actions (as a result of well-behavedness); this is a weaker requirement. Interleaving allows us, for instance, to support the global types on rows “Example 13” and “MP workers” in Table 1.
- Global type $G_1 \bowtie G_2$ specifies the *join* of G_1 and G_2 : every “unconstrained” communication that occurs only in G_1 or only in G_2 is enabled in $G_1 \bowtie G_2$ if, and only if, it is enabled in G_1 or G_2 ; every “constrained” communication that occurs both in G_1 and in G_2 is enabled if, and only if, it is enabled in G_1 and G_2 . See also Example 49, below.
- Global type $[G]_{\gamma_2}^{\gamma_1}$ specifies the *prioritisation* of high-priority γ_1 over low-priority γ_2 in G .

$$\dots \text{ (same as in Figure 5)} \quad \frac{\Xi, \mathcal{Y} \vdash \sum\{O_1, \dots, O_n\} : L \quad \Xi, \mathcal{Y} \vdash \sum\{I_1, \dots, I_m\} : L}{\Xi, \mathcal{Y} \vdash \sum\{O_1, \dots, O_n, I_1, \dots, I_m\} : L}$$

■ **Figure 8** Well-typedness.

$$\begin{array}{c} \dots \text{ (same as in Figure 6a)} \qquad \dots \text{ (same as in Figure 6b)} \\ \frac{G_1 \downarrow \quad G_2 \downarrow}{G_1 ; G_2 \downarrow} \qquad \frac{G_1 \xrightarrow{\gamma} G'_1}{G_1 ; G_2 \xrightarrow{\gamma} G'_1 \cdot G_2} \quad \frac{r(G_1) \cap \{p, q\} = \emptyset \quad G_2 \xrightarrow{pq?t} G'_2}{G_1 ; G_2 \xrightarrow{pq?t} G_1 ; G'_2} \\ \frac{G_1 \downarrow \quad G_2 \downarrow}{G_1 \parallel G_2 \downarrow} \qquad \frac{G_1 \xrightarrow{\gamma} G'_1}{G_1 \parallel G_2 \xrightarrow{\gamma} G'_1 \parallel G_2} \quad \frac{G_2 \xrightarrow{\gamma} G'_2}{G_1 \parallel G_2 \xrightarrow{\gamma} G_1 \parallel G'_2} \\ \frac{G_1 \bowtie G_2 \not\downarrow}{G_1 \bowtie G_2 \downarrow} \quad \frac{G_1 \xrightarrow{\gamma} G'_1 \quad \gamma \notin \mathbf{a}(G_2)}{G_1 \bowtie G_2 \xrightarrow{\gamma} G'_1 \bowtie G_2} \quad \frac{G_2 \xrightarrow{\gamma} G'_2 \quad \gamma \notin \mathbf{a}(G_1)}{G_1 \bowtie G_2 \xrightarrow{\gamma} G_1 \bowtie G'_2} \quad \frac{G_1 \xrightarrow{\gamma} G'_1 \quad G_2 \xrightarrow{\gamma} G'_2}{G_1 \bowtie G_2 \xrightarrow{\gamma} G'_1 \bowtie G'_2} \\ \frac{[G]_{\gamma_2}^{\gamma_1} \not\downarrow}{[G]_{\gamma_2}^{\gamma_1} \downarrow} \quad \frac{G \xrightarrow{\gamma_1} G'}{[G]_{\gamma_2}^{\gamma_1} \xrightarrow{\gamma_1} [G']_{\gamma_2}^{\gamma_1}} \quad \frac{G \xrightarrow{\gamma_2} G' \quad G \xrightarrow{\gamma_1} G'}{[G]_{\gamma_2}^{\gamma_1} \xrightarrow{\gamma_2} [G']_{\gamma_2}^{\gamma_1}} \quad \frac{G \xrightarrow{\gamma} G' \quad \gamma \notin \{\gamma_1, \gamma_2\}}{[G]_{\gamma_2}^{\gamma_1} \xrightarrow{\gamma} [G']_{\gamma_2}^{\gamma_1}} \end{array}$$

(a) Termination (b) Reduction. Let $\mathbf{a}(G) = \{\gamma \mid G \rightarrow \dots \xrightarrow{\gamma}\}$ and $r(G) = \{p, q \mid pq?t \in \mathbf{a}(G)\}$.

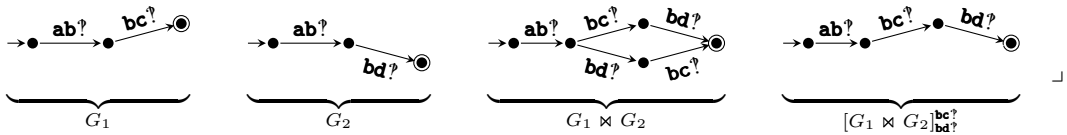
■ **Figure 9** Operational semantics of regular global types, extended.

► **Example 49.** To exemplify join and prioritisation, let $G_1 = \mathbf{a} \rightarrow \mathbf{b} \cdot \mathbf{b} \rightarrow \mathbf{c}$ and $G_2 = \mathbf{a} \rightarrow \mathbf{b} \cdot \mathbf{b} \rightarrow \mathbf{d}$ (data types omitted). Unconstrained are $\mathbf{b} \rightarrow \mathbf{c}$ (only in G_1) and $\mathbf{b} \rightarrow \mathbf{d}$ (only in G_2); constrained is $\mathbf{a} \rightarrow \mathbf{b}$ (both in G_1 and in G_2). Thus, $G_1 \bowtie G_2$ is equivalent to $\mathbf{a} \rightarrow \mathbf{b} \cdot (\mathbf{b} \rightarrow \mathbf{c} \parallel \mathbf{b} \rightarrow \mathbf{d})$: after the constrained communication, the unconstrained communications are interleaved. Thus, $[G_1 \bowtie G_2]_{\mathbf{bd}^?}^{\mathbf{bc}^?}$ is equivalent to $[\mathbf{a} \rightarrow \mathbf{b} \cdot (\mathbf{b} \rightarrow \mathbf{c} \parallel \mathbf{b} \rightarrow \mathbf{d})]_{\mathbf{bd}^?}^{\mathbf{bc}^?}$, which is equivalent to $\mathbf{a} \rightarrow \mathbf{b} \cdot \mathbf{b} \rightarrow \mathbf{c} \cdot \mathbf{b} \rightarrow \mathbf{d}$: after $\mathbf{a} \rightarrow \mathbf{b}$ (no priority), $\mathbf{b} \rightarrow \mathbf{c}$ (high priority) must precede $\mathbf{b} \rightarrow \mathbf{d}$ (low priority). \lrcorner

► **Definition 50** (extension of Definition 43). *See Figure 9.* \lrcorner

- The termination rules for join and prioritisation state that they can terminate when they cannot reduce. This formalises the design decision that operators that constrain the behaviour of operands should be liberal: they should permit as much behaviour as possible within the constraints they impose (avoid premature termination when reductions are still possible); if less behaviour is required, more constraints can always be imposed.
- The second reduction rule for weak sequencing states that G_2 can start reducing before G_1 has finished reducing, when the roles that participate in the reduction of G_2 are disjoint from those that participate in reductions of G_1 (i.e., these reductions are independent).
- The first reduction rule (resp. second) for join states that it can γ -reduce when G_1 (resp. G_2) can, now, but G_2 (resp. G_1) cannot, ever (i.e., γ is unconstrained). The third reduction rule states that it can γ -reduce when G_1 and G_2 can (i.e., γ is constrained).
- The first reduction rule for prioritisation states that it can γ_1 -reduce (high priority) when G can. The second reduction rule states that it can γ_2 -reduce (low priority) when it cannot γ_1 -reduce. The third reduction rule states that it can γ -reduce when G can.

► **Example 51.** The following graphs visualise the operational semantics for Example 49:



We note that an evaluation of the usefulness of the added operators, as practical language primitives, is not really part of the present scope; here, our only aim was to give an impression of the future potential of the new techniques. Other possible global type primitives that may deserve future consideration include delayed choice [5], roles-as-ports composition [47], stateful global types (cf. stateful choreographies [28]), operators for higher-order protocols, and syntax for general models of behaviour (as mentioned towards the end of §1.2).

5 Related Work

This paper contributes to a line of research to increase the expressiveness of MPST [39]. Regarding basic features, previous works have focussed on two limitations of *directed choice* of the form $\sum\{p \rightarrow q : t_i . G_i\}_{1 \leq i \leq n}$: **(1)** every branch must start with the same sender and the same receiver as every other branch; **(2)** every “third role” that does not participate in the first communication of every branch must have the same behaviour in every branch.

- **Merging:** Honda et al. address limitation **2** by allowing “third roles” to have different behaviour in different branches when they are timely informed of the chosen branch [18]. The approach relies on a function to syntactically *merge* local types; it is adopted by many (e.g., [25, 33, 57]), but shown to be brittle [62]. In contrast, using our new projection and type checking techniques, we address limitation **2** without merging (Remark 10). Another approach that addresses limitation **2** without merging was developed by Scalas–Yoshida [62]. It works in three steps: first, every local type is interpreted as an automaton that specifies one role alone (similar to our operational semantics of implicit local types); next, the automata are composed into a product automaton – exponentially sized in the worse case – that specifies all roles together; last, the product automaton is checked for satisfaction of a special temporal logic formula φ , which entails type soundness. However, this method is non-compositional: the premise of the typing rule for families of processes (which depends on satisfaction of φ) cannot be checked separately for every role. Such non-compositional approaches to MPST have already been shown to have scalability issues [44]. Conversely, our typing rule for families is fully compositional (Remark 27).
- **Located/mixed choice:** Several teams of authors address limitation **1** by allowing every branch to start with a different receiver than every other branch. In earlier works that support such *located choice* of the form $\sum\{p \rightarrow q_i : t_i . G_i\}_{1 \leq i \leq n}$, communication races in the continuations are forbidden [9, 20, 31, 42, 51]; in later works, they are allowed [21–23, 54, 65]. We support them, too. However, the authors of these papers prove theorems for a closed set of global type operators, including $\sum\{p \rightarrow q_i : t_i . G_i\}_{1 \leq i \leq n}$. Instead, we prove theorems for an open set of global type operators, as demonstrated in §2.2 and §4. Verification of mixed input/output processes using session typing is a long-standing open problem. Progress was made by Casal et al. [19] (binary), Kouzapas–Yoshida (multiparty, but unpublished so far [69, ref. 24]), and Jongmans–Yoshida [44] (multiparty, but no type checking). We can verify multiparty non-deterministic mixed input/output processes for the first time (but not yet deterministic mixed choice), as demonstrated in §2.2.

The usage of the operational semantics of local types was first studied in the context of *multiparty compatibility* [32] and extensions [8, 51, 52]. The idea is to interpret local types as *communicating finite state machines* (CFSM) [11]. Multiparty compatibility, then, is a predicate on the joint state space of the CFSMs to ensure safety and liveness. As such, a key difference between multiparty compatibility (MC) and this paper’s well-behavedness (WB) is that MC is non-compositional (i.e., the joint state space must be computed, so MC cannot

be checked separately for every role), whereas WB is fully compositional (Remark 27). A rudimentary version of WB was studied by Jongmans–Yoshida [44], but it is less expressive (e.g., they do not support Example 15) and limited to projection (no type checking). A version of WB for global types was studied by Gheri et al. [34], in the context of choreography automata [6], but it is limited to projection (no type checking).

There are several non-traditional techniques for projection in the MPST literature. Lopez et al. [53] capture projection in a decidable type equivalence. Castellani et al. [22] and Hamers et al. [37] do not use projection at all, but type-check families of processes against global types (non-compositional). Last, the concept of implicit local types in this paper generalises an idea by Van Glabbeek et al. [65], who define merging as a local type operator.

6 Conclusion

We introduced two new techniques to significantly improve the expressiveness of the MPST method: projection is based on *implicit* local types instead of explicit; type checking is based on the *operational semantics* of implicit local types instead of on the syntax. Classes of protocols that can now be specified/implemented/verified for the first time using the MPST method include: recursive protocols in which different roles participate in different branches (Example 2, Example 3); protocols in which a receiver chooses the sender of the first communication (Example 6, Example 8, Example 9); protocols in which multiple roles synchronously choose both the sender and the receiver of a next communication (Example 5, Example 6), implemented as mixed input/output processes. We presented the theory of the new techniques, as well as their future potential, and we demonstrated their present capabilities to effectively support regular expressions as global types (not possible before).

As evidence that the new techniques are implementable, we implemented them; this implementation is available as a companion artefact (published in DARTS).

We aim to push the new techniques of this paper forward towards a new branch of research in MPST, centred around operational semantics of local types in typing rules; incidentally, it could be a natural path to explore for behavioural typing in general, too. In particular, we are keen to apply the new techniques for projection and type checking also to *asynchronous communication* and *parametrised protocols/indexed roles* [25, 33]. In both cases, the main challenge is how to ensure decidability (keep the reduction relations finite).

References

- 1 Luca Aceto, Anna Ingólfssdóttir, and Jiri Srba. The algorithmics of bisimilarity. In *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge tracts in theoretical computer science*, pages 100–172. Cambridge University Press, 2012.
- 2 Bernardo Almeida, Andreia Mordido, Peter Thiemann, and Vasco T. Vasconcelos. Polymorphic lambda calculus with context-free session types. *Inf. Comput.*, 289(Part):104948, 2022.
- 3 Bernardo Almeida, Andreia Mordido, and Vasco T. Vasconcelos. Deciding the bisimilarity of context-free session types. In *TACAS (2)*, volume 12079 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2020.
- 4 Jos C. M. Baeten, Flavio Corradini, and Clemens Grabmayer. A characterization of regular expressions under bisimulation. *J. ACM*, 54(2):6, 2007.
- 5 Jos C. M. Baeten and Sjouke Mauw. Delayed choice: an operator for joining message sequence charts. In *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 340–354. Chapman & Hall, 1994.
- 6 Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020.

- 7 J. F. A. K. Van Benthem. Hintikka on analyticity. *Journal of Philosophical Logic*, 3(4):419–431, 1974. doi:10.1007/bf00257484.
- 8 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, volume 42 of *LIPICs*, pages 283–296. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
- 9 Laura Bocchi, Hernán C. Melgratti, and Emilio Tuosto. Resolving non-determinism in choreographies. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 493–512. Springer, 2014.
- 10 Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In *CONCUR*, volume 8704 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2014.
- 11 Daniel Brand and Pitro Zafropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
- 12 Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *SC@ETAPS*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007.
- 13 Mario Bravetti and Gianluigi Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In *WS-FM*, volume 5387 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2008.
- 14 Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration conformance for system design. In *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2006.
- 15 Sara Capecchi, Iliaria Castellani, and Mariangiola Dezani-Ciancaglini. Typing access control and secure information flow in sessions. *Inf. Comput.*, 238:68–105, 2014.
- 16 Sara Capecchi, Iliaria Castellani, and Mariangiola Dezani-Ciancaglini. Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science*, 26(8):1352–1394, 2016.
- 17 Sara Capecchi, Iliaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. Session types for access and information flow control. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2010.
- 18 Marco Carbone, Nobuko Yoshida, and Kohei Honda. Asynchronous session types: Exceptions and multiparty interactions. In *SFM*, volume 5569 of *Lecture Notes in Computer Science*, pages 187–212. Springer, 2009.
- 19 Filipe Casal, Andreia Mordido, and Vasco T. Vasconcelos. Mixed sessions. *Theor. Comput. Sci.*, 897:23–48, 2022.
- 20 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, 8(1), 2012.
- 21 Iliaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Reversible sessions with flexible choices. *Acta Informatica*, 56(7-8):553–583, 2019.
- 22 Iliaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Asynchronous sessions with input races. *CoRR*, abs/2203.12876, 2022.
- 23 Iliaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. Global types with internal delegation. *Theor. Comput. Sci.*, 807:128–153, 2020.
- 24 Iliaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Asp. Comput.*, 28(4):669–696, 2016.
- 25 David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *PACMPL*, 3(POPL):29:1–29:30, 2019.
- 26 Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. API generation for multiparty session types, revisited and revised using scala 3. In *ECOOOP*, volume 222 of *LIPICs*, pages 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- 27 Alex Coto, Roberto Guanciale, and Emilio Tuosto. An abstract framework for choreographic testing. *J. Log. Algebraic Methods Program.*, 123:100712, 2021.

- 28 Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. The paths to choreography extraction. In *FoSSaCS*, volume 10203 of *Lecture Notes in Computer Science*, pages 424–440, 2017.
- 29 Ugo de'Liguoro, Hernán C. Melgratti, and Emilio Tuosto. Towards refinable choreographies. *J. Log. Algebraic Methods Program.*, 127:100776, 2022.
- 30 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011.
- 31 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012.
- 32 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013.
- 33 Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. *Logical Methods in Computer Science*, 8(4), 2012.
- 34 Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-by-contract for flexible multiparty session protocols. In *ECOOP*, volume 222 of *LIPICs*, pages 8:1–8:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- 35 Roberto Guanciale and Emilio Tuosto. Realisability of pomsets. *J. Log. Algebraic Methods Program.*, 108:69–89, 2019.
- 36 Ruben Hamers, Erik Horlings, and Sung-Shik Jongmans. The discourje project: run-time verification of communication protocols in clojure. *Int. J. Softw. Tools Technol. Transf.*, 24(5):757–782, 2022.
- 37 Ruben Hamers and Sung-Shik Jongmans. Discourje: Runtime verification of communication protocols in clojure. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 2020.
- 38 Jaakko Hintikka. *Logic, Language-Games and Information: Kantian Themes in the Philosophy of Logic*. Oxford, England: Oxford, Clarendon Press, 1973.
- 39 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.
- 40 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.
- 41 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2016.
- 42 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017.
- 43 Sung-Shik Jongmans and Francisco Ferreira. Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types (Technical Report). Technical Report OUNL-CS-2023-01, Open University of the Netherlands, 2023.
- 44 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 251–279. Springer, 2020.
- 45 Alex C. Keizer, Henning Basold, and Jorge A. Pérez. Session coalgebras: A coalgebraic view on regular and context-free session types. *ACM Trans. Program. Lang. Syst.*, 44(3):18:1–18:45, 2022.
- 46 Jonathan King, Nicholas Ng, and Nobuko Yoshida. Multiparty session type-safe web development with static linearity. In *PLACES@ETAPS*, volume 291 of *EPTCS*, pages 35–46, 2019.
- 47 Christian Koehler and Dave Clarke. Decomposing port automata. In *SAC*, pages 1369–1373. ACM, 2009.

- 48 Nicolas Lagaillardie, Romyana Neykova, and Nobuko Yoshida. Implementing multiparty session types in rust. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2020.
- 49 Nicolas Lagaillardie, Romyana Neykova, and Nobuko Yoshida. Stay safe under panic: Affine rust programming with multiparty session types. In *ECOOP*, volume 222 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- 50 Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM*, pages 323–332. IEEE Computer Society, 2008.
- 51 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232. ACM, 2015.
- 52 Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In *CAV (1)*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019.
- 53 Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Protocol-based verification of message-passing parallel programs. In *OOPSLA*, pages 280–298. ACM, 2015.
- 54 Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. Generalising projection in asynchronous multiparty session types. In *CONCUR*, volume 203 of *LIPIcs*, pages 35:1–35:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- 55 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
- 56 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in typescript with routed multiparty session types. In *CC*, pages 94–106. ACM, 2021.
- 57 Romyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.
- 58 Romyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in $f\#$. In *CC*, pages 128–138. ACM, 2018.
- 59 Nicholas Ng and Nobuko Yoshida. Pabble: parameterised scribble. *Service Oriented Computing and Applications*, 9(3-4):269–284, 2015.
- 60 Luca Padovani. Context-free session type inference. *ACM Trans. Program. Lang. Syst.*, 41(2):9:1–9:37, 2019.
- 61 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
- 62 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019.
- 63 Peter Thiemann and Vasco T. Vasconcelos. Context-free session types. In *ICFP*, pages 462–475. ACM, 2016.
- 64 Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies. *J. Log. Algebraic Methods Program.*, 95:17–40, 2018.
- 65 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming just enough fairness to make session types complete for lock-freedom. In *LICS*, pages 1–13. IEEE, 2021.
- 66 Rob J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.
- 67 Rob J. van Glabbeek. The linear time - branching time spectrum II. In *CONCUR*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.
- 68 Rob J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.

42:30 Sound, Regular Multiparty Sessions via Implicit Local Types

- 69 Vasco T. Vasconcelos, Filipe Casal, Bernardo Almeida, and Andreia Mordido. Mixed sessions. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 715–742. Springer, 2020.
- 70 Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In *FCT*, volume 12867 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2021.
- 71 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Romyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020.

On the Rise of Modern Software Documentation

Marco Raglianti ✉ 

REVEAL @ Software Institute – USI, Lugano, Switzerland

Csaba Nagy ✉ 

REVEAL @ Software Institute – USI, Lugano, Switzerland

Roberto Minelli ✉ 

REVEAL @ Software Institute – USI, Lugano, Switzerland

Bin Lin ✉ 

Radboud University, Nijmegen, The Netherlands

Michele Lanza ✉ 

REVEAL @ Software Institute – USI, Lugano, Switzerland

Abstract

Classical software documentation, as it was conceived and intended decades ago, is not the only reality anymore. Official documentation from authoritative and official sources is being replaced by real-time collaborative platforms and ecosystems that have seen a surge, influenced by changes in society, technology, and best practices. These modern tools influence the way developers document the conception, design, and implementation of software. As a by-product of these shifts, developers are changing their way of communicating about software. Where once official documentation stood as the only truth about a project, we now find a multitude of volatile and heterogeneous documentation sources, forming a complex and ever-changing *documentation landscape*.

Software projects often include a top-level README file with important information, which we leverage to identify their documentation landscape. Starting from ~12K GitHub repositories, we mine their README files to extract links to additional documentation sources. We present a qualitative analysis, revealing multiple dimensions of the documentation landscape (e.g., content type, source type), highlighting important insights. By analyzing instant messaging application links (e.g., Gitter, Slack, Discord) in the histories of README files, we show how this part of the landscape has grown and evolved in the last decade.

Our findings show that modern documentation encompasses communication platforms, which are exploding in popularity. This is not a passing phenomenon: On the contrary, it entails a number of unknowns and socio-technical problems the research community is currently ill-prepared to tackle.

2012 ACM Subject Classification Software and its engineering → Collaboration in software development; Human-centered computing → Collaborative and social computing

Keywords and phrases software documentation landscape, GitHub README, instant messaging

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.43

Category Pearl/Brave New Idea

Supplementary Material *Software*: <https://figshare.com/s/33c8af534dba61d72c41>

Funding This work is supported by the Swiss National Science Foundation (SNSF) through the project “INSTINCT” (SNF Project No. 190113).

Acknowledgements Marco Raglianti would also like to thank the Swiss Group for Original and Outside-the-box Software Engineering (CHOOSE) for sponsoring the trip to the conference.

1 Introduction

Times are changing. This is even more true for software engineering. Major shifts have occurred, induced by the emergence of platforms like GitHub and StackOverflow, fundamentally changing how developers (and users) communicate about software projects: Mailing lists and forums are declining in favor of multi-media instant messaging platforms, such as Gitter, Slack, Discord, and GitHub Discussions, e.g., [9, 18, 27, 30, 33, 45, 46, 49, 50, 56, 58, 66, 67].



© Marco Raglianti, Csaba Nagy, Roberto Minelli, Bin Lin, and Michele Lanza;
licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 43; pp. 43:1–43:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Software documentation, a critical asset for developers [3], has been studied extensively with respect to its quality and usefulness [4, 10, 14, 19, 21, 54, 61, 79]. Nevertheless, the impact of the subtle but constant drift induced by new platforms is still to be evaluated. What are the implications for program comprehension if a tweet can influence how developers treat a bug [38]? Can the tweets on the usage of an API also serve as documentation? Classical software documentation, as we have known it, is being replaced by “communication”.

Documentation went from a clunky, and rather unloved, endeavor to becoming a fast-paced and volatile side dish. The utopia of “*on-demand documentation*” by Robillard et al. [55], is being replaced by a dystopia of an ever-changing landscape; documentation is waved away with sentences like “*check Discord*” or “*it’s in the pull request comments.*”

This change is more than just cosmetic, it is considerably affected by the richness of new media, influencing the cognitive processes that underlie communication [53]. Modern media-rich platforms offer vastly different mechanisms which are simply not there in classical electronic communication means. Moreover, developers do not only hold ephemeral discussions that they must be able to access now. They share knowledge (e.g., code examples, screenshots, howtos) that is important for them in the future, and they do not have (or rather: take) the time to persist it in a classical software documentation form (e.g., Wiki). Instant messaging is just too enticing for that. But, they will need long-term access to this knowledge and want to keep it searchable¹ [20] and organizable² [44]. They choose their platforms accordingly, for example, avoiding limitations in retrievable history [2], and are willing to pay significant sums for such services [12].

As the cards on documentation are being reshuffled, things seem murky: What happens to the body of knowledge contained in the repositories of classical communication platforms? What is the impact on standard software documentation? How do developers use modern platforms, and what does this imply?

The Spectrum Example. *Spectrum, a multi-forum community hosting platform, was hosting dozens of software related communities about frameworks (e.g., React, Laravel), UI design (e.g., Figma), front-end coding (e.g., CodePen), and developers’ networks in general (e.g., SpecFM). On Aug 24, 2021, to preserve history while pushing forward the adoption of new communication infrastructures, it was announced that “the time has come for the planned archival of Spectrum to focus our efforts on GitHub Discussions” [36]. Spectrum has become “read-only – no 404s or lost internet history.” The Spectrum team acknowledged the importance of conversations held on the platform and tried to avoid the limitations of relying on the Internet Archive for preservation [71]. Many Spectrum communities had already moved to GitHub Discussions, for reliability and flexibility reasons: Having code and the community in the same place outweighed other factors in the decision to change.*




We present an overview of the *documentation landscape* (i.e., a map of potential documentation sources) emerging from the analysis of ~12K GitHub projects. We explore current trends in documentation platforms and the relationship between documentation and communication platforms, exemplified by the tendency in a project’s README to include the latter as an indirect source of the former.

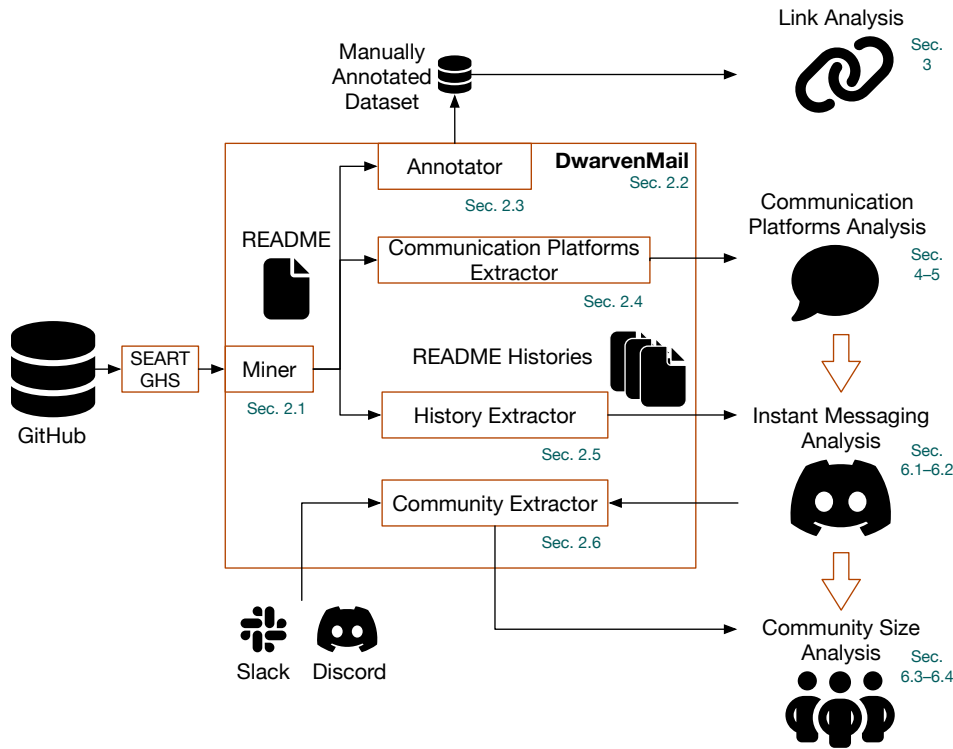
We show the most representative values in different dimensions characterizing the landscape. We then proceed more in-depth with the history of modern communication platforms. We show how some platforms have seen increasing adoption, reached a plateau, and finally started their decline. Our analysis provides insights into the many implications of this

¹ Especially for large communities, without limitations, as reported in this blog post.

² As demonstrated by the presence of an ecosystem built on top of instant messaging applications.

ongoing phenomenon for software documentation. Finally, we discuss possible features that future platforms should have to mitigate some of the perils introduced by these continuous shifts. We use the following icons to highlight salient points:

 Insight  Idea/Future Work  Threat



■ **Figure 1** DWARVENMAIL and Analyses Overview.

2 Dataset Creation and DWARVENMAIL

This section details the procedure and tool support (DWARVENMAIL) we implemented to collect the data for our analyses (Figure 1). We present the initial dataset, the mining procedure, our manual annotation, and details about the individual analyses we performed.

2.1 Project Mining

Starting from all repositories currently hosted on GitHub we used SEART-GHS [13] to compose a relevant dataset, applying the following filtering criteria: at least 2,000 commits (i.e., to eliminate toy projects), more than 10 contributors (i.e., to ensure that a certain number of people need to interact with each other to tackle the development effort), and more than 100 stars (i.e., to ensure that the projects are relevant to at least a handful of people). We only considered projects created before July 1, 2022 and excluded forks [22]. SEART-GHS currently monitors about 1.2M GitHub repositories. Projects excluded in SEART-GHS for having less than 10 stars [13] would have been excluded by the more restrictive criterion we applied, removing projects with less than 100 stars.



Filtering based on the number of stars might not be sufficient to select relevant projects. Nevertheless, starring can be important for project developers and managers [8]. We used this criterion as a common method for filtering out toy projects in GitHub (e.g., see [80]).

We performed the filtering on July 14, 2022, resulting in 12,461 projects exported as JSON input for DWARVENMAIL. After removing 374 aliases and 6 renamed forks, the final scraped dataset consists of 12,081 projects.

Table 1 presents an overview of the projects according to their languages. The *All* column shows the total number of projects, the *CP* column the projects where we could identify communication platforms (see Section 2.4), and the *IM* column the projects with instant messaging platforms. Percentages are derived with respect to the *All* column, while Δ percentages are relative to the *Total* row percentages.

Overall, 57.3% of the projects we analyzed feature at least one communication platform. An interesting observation is that systems written in “lower level / traditional” languages (C, PHP, Shell) tend to be below the overall average, while systems written in more “modern” languages (C#, Go, Rust, TypeScript) are more inclined to feature communication platforms. The difference is even more evident for recently popularized languages if we consider projects with instant messaging platforms (e.g., Go and Rust increase from +8.1% to +11.6% and from +9.5% to +16.7% respectively).

We performed multiple *One Proportion Z-Tests* (one for each language) and the difference in the proportion of projects using communication platforms for each language (r) and the overall dataset (r_0) is statistically significant for C, C#, Go, PHP, Rust, Shell, and TypeScript ($H_0 : r = r_0$, two-tailed Bonferroni corrected p-value < 0.0038). The same results hold for projects with instant messaging platforms.

■ **Table 1** Projects and Represented Languages.

Language	Projects						
	All	CP	CP %	$\Delta_{CP}\%$	IM	IM %	$\Delta_{IM}\%$
C	1,240	548	44.2%	-13.1%	248	20.0%	-9.2%
C#	543	378	69.6%	+12.3%	214	39.4%	+10.2%
C++	1,707	926	54.2%	-3.1%	469	27.5%	-1.7%
Go	677	443	65.4%	+8.1%	276	40.8%	+11.6%
Java	1,510	860	57.0%	-0.4%	432	28.6%	-0.6%
JavaScript	1,528	899	58.8%	+1.5%	440	28.8%	-0.4%
PHP	733	379	51.7%	-5.6%	165	22.5%	-6.7%
Python	1,806	1,094	60.6%	+3.3%	557	30.8%	+1.7%
Ruby	406	238	58.6%	+1.3%	103	25.4%	-3.8%
Rust	244	163	66.8%	+9.5%	112	45.9%	+16.7%
Shell	205	93	45.4%	-11.9%	35	17.1%	-12.1%
TypeScript	895	575	64.2%	+6.9%	314	35.1%	+5.9%
Other / Unspecified	587	328	55.9%	-1.4%	160	27.3%	-1.9%
Total	12,081	6,924	57.3%		3,525	29.2%	

2.2 Tool Support: DWARVENMAIL

To support our analyses, we developed DWARVENMAIL, a Python application to scrape GitHub and extract information about projects’ README files and their history. It features an object-oriented domain model to facilitate the extraction of insights from exploration.

DWARVENMAIL also supports manual inspection, link extraction, and classification from README files (see Section 2.3). DWARVENMAIL takes the list of projects in the dataset and uses the REST API of GitHub and web scraping [81] to extract the information needed to build its internal domain model. It is implemented as a multiprocessing application to speed up the scraping. Each process uses a different API key to access GitHub in parallel through

PyGitHub [23]. Parallelization is handled at project level: Each process gets a project from a queue and starts to fetch the data. Processes are also responsible for not exceeding GitHub rate limits associated with their API keys.

2.3 Manual Annotation

To examine the documentation sources and communication platforms of the projects, we performed a qualitative analysis of their README files. We relied on open card sorting, a well-established method for knowledge elicitation and classification [7, 41, 63, 76, 77], to incrementally refine the list of possible sources with flexible categories.

We manually reviewed the README files of the projects, extracted their links to documentation sources and organized them into categories. Given the considerable effort needed to annotate README files manually, we opted for a saturation approach [57]. We started with a sample set of 35 projects selected through stratified sampling, ensuring a balanced distribution among programming languages.

Two authors independently annotated each project README. Then we repeated the process in subsequent batches with 5 projects per batch until no new labels were added in two consecutive batches. We reached saturation after annotating 60 projects. In the end, we discussed conflicts and merged categories where needed. The process resulted in 2,349 links with 282 link types, which we discuss in Section 3. The creation of manually annotated datasets was supported by the *Annotator* module of DWARVENMAIL (Figure 2).

The screenshot shows the DWARVENMAIL Annotator interface. It features two side-by-side panes: 'Raw' and 'Rendered'. The 'Raw' pane displays the original Markdown text of a README file for 'The Bug Genie', including build status badges, social media links, and a list of features. The 'Rendered' pane shows a preview of the README as it would appear on GitHub, with the same content rendered into HTML. Below the panes, there are controls for adding and managing links, including a 'Link' input field, dropdown menus for 'Link Format', 'Content Type', 'Source Type', and 'Source Instance', and a table of existing links.

Remove	Single Type	Link Format	Content Type	Source Type	Source Instance	Link
<input type="checkbox"/>	Badge	CI / CD > Testing		Third Party Service	Travis	https://travis-ci.org/thebuggenie/thebuggenie
<input type="checkbox"/>	Badge	General Community Hub		Instant Messaging	Gitter	https://gitter.im/thebuggenie/general

■ **Figure 2** DWARVENMAIL Annotator – Project Annotation Page.

An annotator ran the Flask application locally, pulled from Git the latest updates by other annotators, started a batch of annotations, committed, and pushed the modified files.

The Annotator's homepage shows a list of projects to annotate and the annotation status (i.e., who annotated what). Selecting a project opens the project annotation page (Figure 2) where one can browse the README of the selected project.

The project annotation page uses two side-by-side panes to present the README. The left one represents the raw Markdown version of the README. The right pane shows a partially rendered version (i.e., similar to what a user sees on GitHub).

2.4 Parsing Links: Strategy & Heuristics

We performed a quantitative analysis of communication platforms in README files (Sections 4 and 5). To support automatic platform extraction in such a large number of projects we used an approach based on Regular Expressions (RE).

For each communication platform that we discovered, we devised REs that would match the link as closely as possible, while retaining sufficient generality to abstract specific aspects (e.g., project name, internet domain, optional protocol). Possibly more than one RE has been associated with each platform. To fine-tune the REs, DWARVENMAIL features a detailed log generation for manual inspection of candidate and invalid links during the refinement.

DWARVENMAIL parses all the links in a README according to the set of identified REs. When a link is found, specific exclusion criteria are applied. A set of rules removes links to images, badge icons, platforms' generic homepages, and partial or invalid URLs (e.g., shorthands for Markdown sections captured by the REs).

The remaining links are normalized in a standard format and duplicates are removed. Platforms not directly linked in the README (e.g., collected in a list on the *Community* page of the project website) are omitted by the employed scraping algorithm. To reduce the false positive rate, DWARVENMAIL also verifies that links point to valid web pages (i.e., the server does not respond with an HTTP 404 Not Found). After this refinement, we obtain the final set of communication platforms referenced by the project READMEs.

2.5 Parsing README Histories

For projects referencing Gitter, Slack, and Discord as communication platforms, we analyzed the history of their READMEs to discover when those platforms appeared for the first time. In this case, the approach outlined above to exclude invalid links (Section 2.4) would not produce the desired results, because a link that is not valid today could have been valid in the past. This cannot be checked without an archive, or historical information. Hence, in our approach we assume that links with proper format were valid in the past. To reduce false positives, we used the most specific format able to capture the link.

2.6 Community Size

We include the Discord and Slack community size (i.e., number of members) in our domain model. The most popular way to add people to a Discord server is through an *invite link* [15]. Clicking on an invite link, brings the user to a page with metadata about the server (e.g., number of total members, number of online members). We gathered Discord community sizes by scraping the data from these invite pages. In the case of Slack, only 15% of the projects in our dataset have information about the community size on the invite page.



More effort is needed to explore communities that do not conform to a standard and/or customize their invite link to pursue specific goals (e.g., authorization workflow, authentication, spam prevention, analytics).

Extending the percentage of projects whose community size is correctly scraped could improve the reliability of results discussed in Sections 6.3 and 6.4.

2.7 Data Availability and Replication Package

We provide a replication package, publicly available on Figshare [51], containing the source code of DWARVENMAIL, the input dataset, the manually annotated projects, the serialized domain model of the scraped dataset, charts and tables exported from DWARVENMAIL.

3 Documentation Landscape

We define the documentation landscape of a software system as all the possible sources of information able to support design, implementation, comprehension, maintenance, and evolution of the system. Software documentation is a fundamental asset for developers and practitioners [3], when it is correct and up-to-date [10, 14, 19, 21, 54, 61], with its costs and benefits [79]. Modern software documentation is an ever expanding field. New sources include blogs [43], Twitter [72], StackOverflow [47], instant messaging applications [9, 18, 30, 33, 45, 46, 49, 50, 56, 58, 66, 67], news aggregators [6], and forums [27].

GitHub README files in Markdown (.md) format are a good starting point for a project from where all relevant documentation should be reachable.

Documentation sources in README files can either be directly referred to or behind multiple steps of indirection. An example of the former case is an invitation link that can be copy/pasted directly in Discord to access the community server of the project. In the latter case, the README could point to a community web page which in turn contains links to the mailing list, a Slack channel for Q&A, and potentially other communication and documentation sources.

The manual annotation presented in Section 2.3 produced 282 *single type* link tags. The links can come in many flavors thanks to the markdown format, ranging from pure textual hyperlinks to badges and images that link to external resources. We inspected them and identified three key dimensions of the documentation landscape: *content type*, *source type*, and *source instance*. We split single type tags into these three dimensions. We analyzed examples of each link type to disambiguate or enrich the classification when the original annotation had missing information.

Table 2 shows the top-15 most representative values for each dimension. The complete list of tags is available in the replication package [51].



The three key dimensions we propose to describe the documentation landscape of a software system are content type, source type, and source instance, exemplified as links in GitHub READMEs.

Source type, source instance, and content type could describe a link like: “This link is in the form of a *Badge*, it points to a *Wiki* on *Travis.com*, and contains information related to *CI / CD*.” Each dimension is instantiated with one of the possible tags for that category, forming a signature of the documentation source pointed by the link.



Exploring these dimensions could improve the automatic extraction of links and their features, to characterize and understand the (evolution of the) documentation landscape.

Link format. Link formats come in many flavors, also due to the fact that markdown files, while being textual, are usually inspected using a (multimedia capable) web browser. Badges, for example, are very common in GitHub README files, used to convey imminent information through iconic representation of a summary of the pointed resource (e.g., build status passing) where the link itself allows, if followed, to reach more extensive information (e.g., build process report). Masked links are another common practice to add links (not exclusively) to markdown documents.



Not all links in a raw README file are human readable links in the rendered README.

Content type. This is the primary dimension of the documentation landscape, denoting **what** kind of information is present in the landscape. There is a smooth gradient in content types regarding the number of links, but it is worth noting that there is no “standard”, but

■ **Table 2** Top-15 Most Relevant Tags, Number of Projects, and Links for Each Dimension. The percentage indicates the ratio of projects containing at least one link with the specified tag.

(a) Content Type.

Projects	Content Type	Links
36 (60%)	General Community Hub	141
29 (48%)	Official Documentation	97
28 (47%)	License	68
25 (42%)	Contributing	56
23 (38%)	Issues	52
23 (38%)	CI/CD	50
21 (35%)	Project Repository	76
20 (33%)	Relevant Projects	132
20 (33%)	Dependency/Environment	84
19 (32%)	Releases	60
16 (27%)	In-Repository Resource	67
16 (27%)	Package Repository	47
14 (23%)	CI/CD > Testing	24
13 (22%)	Installation Instructions	24
11 (18%)	Code Coverage	22

(b) Source Type.

Projects	Source Type	Links
55 (92%)	Homepage/Website	436
41 (68%)	Collaborative Platform	188
36 (60%)	Third Party Service	169
34 (57%)	Wiki	125
32 (53%)	Repository	166
28 (47%)	Sourcefile/Sourcefolder	151
25 (42%)	Instant Messaging	84
11 (18%)	Auxiliary README	21
10 (17%)	Readme Section/Anchor	44
9 (15%)	Mailing List	27
9 (15%)	Forum	20
8 (13%)	Image/GIF	21
8 (13%)	Blog	20
7 (12%)	Email Address	12
6 (10%)	Video	13

(c) Source Instance.

Projects	Source Instance	Links
33 (55%)	GitHub	179
16 (27%)	GitHub Workflows	50
13 (22%)	GitHub Releases	27
12 (20%)	Travis	22
11 (18%)	Gitter	34
9 (15%)	Google Groups	24
7 (12%)	Discord	18
7 (12%)	Codecov	14
7 (12%)	Python Package Index	13
6 (10%)	Twitter	12
6 (10%)	StackOverflow	9
5 (8%)	Slack	18
5 (8%)	Maven	11
5 (8%)	Read the Docs	5
4 (7%)	GitHub Profile	108

rather project-specific landscapes. Most relevant are *general community hubs*: Discord servers, Slack workspaces, Gitter rooms, IRC channels, mailing lists, and forums, with their internal structure for different topics, dedicated to a general community of users and practitioners.



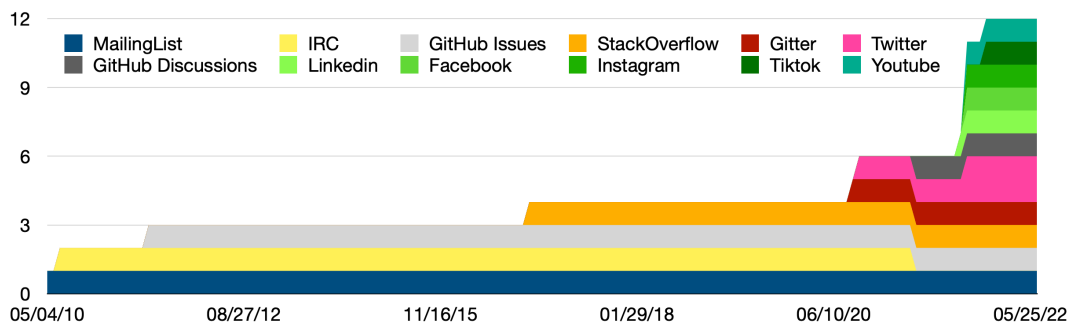
Content type is relevant to interpret what a piece of documentation is about. There is no standard to the documentation landscape, each project develops its own. Even the top content types (community hubs, official documentation) are present in only half of the projects.

Source type. The *source type* dimension refers to the format of the content at the link's destination. This dimension is relevant for automatically extracting the documentation landscape since it determines how the content can be retrieved and parsed. *Homepage / websites*, the most relevant source type by a large margin, can be scraped with traditional web scraping techniques. *Collaborative platforms* like GitHub and Bugzilla could be addressed via their custom APIs. *Image / GIF > Screenshots*, further down in terms of relevance, would benefit from image segmentation and analysis approaches to extract, for example, documented user interface features. We also notice that links to mailing lists are fewer than those to IM applications, a trend we analyze in more detail in Section 4.



Source type captures *how* documentation is presented and how it can be accessed. Almost all projects feature a head quarters website, i.e., the go-to place to learn about a project. These starting points are then often complemented by a plethora of other sources, ranging from Wikis to forums and instant messaging platforms.

When analyzing the evolution through time of a README file we detect in many cases that the source types come and go, inducing “tectonic movements” in the landscape, as we can observe in the example depicted in Figure 3.



■ **Figure 3** Evolution of Communication Platforms in the “Scikit-learn” Project.

The Scikit-learn project, born in 2010, sees a mailing list as its initial documentation landscape, complemented shortly after by an IRC channel (which stopped existing a decade later). GitHub Issues is added within the project’s first year, while StackOverflow becomes part of the landscape in 2017. It is within the past 2 years that the landscape experiences an earthquake, with many new sources appearing, while the IRC channel is removed (it is worth noting that IRC and its successor, Gitter, co-exist for a year). At the time of writing the project in question features 11 different sources.



The documentation landscape of projects evolves together with the project. Especially in the past few years the source types have exploded in number, rendering the landscape highly dispersive.



The fact that there are more sources does not imply that the overall documentation of the system is better, on the contrary: We have observed an overall trend toward more volatile sources, mostly due to the rise of instant multimedia messaging platforms.

Source instance. The third dimension is a derivate of source type. For each type we can have multiple possible source instances, usually of a competing nature (see Section 5) with a similar purpose. Rather unsurprisingly for GitHub projects, GitHub itself with related instances of profiles, workflows, releases, and instant messaging (i.e., Gitter) takes top three, the 5th, and the 15th places. Services for package repositories (e.g., Python Package Index [48], Maven [62]) and CI/CD (e.g., Travis CI [73], Codecov [11]), messaging applications like Slack [60]/Discord [16], and also articles on the Medium platform [1] represent interesting research avenues.



Source instance can be seen as *where* (or by whom) documentation is “hosted.”



Source instances vary wildly, and new players constantly enter the stage. For example, recent changes in the pricing model of Slack might have influenced the ongoing mass migration toward other instant messaging platforms, of which there are dozens, with Discord quickly becoming the preferred alternative.



Tags in the three dimensions appear in different combinations, not all equally likely. Further research on the most common patterns could shed light on form and content interplay in software documentation.

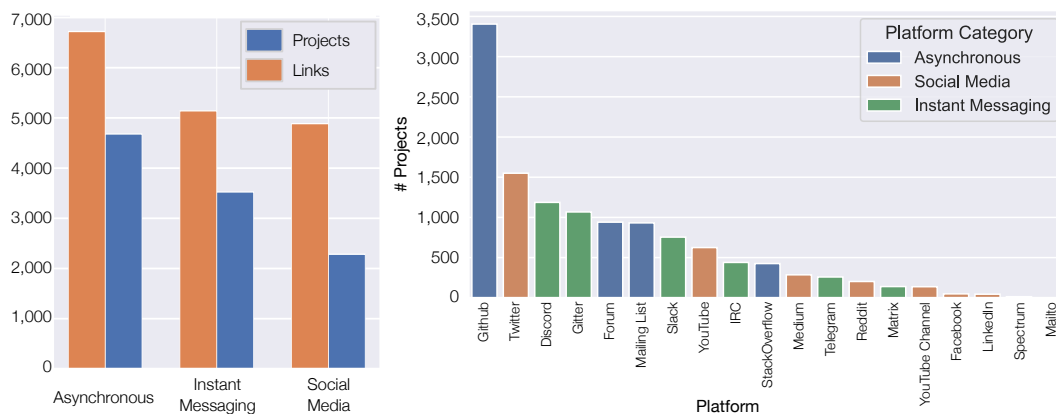
4 Modern Communication Platforms

One of the recent major shifts in software development has been the emergence of various multimedia instant messaging platforms, such as Slack [60], Discord [16], and Gitter [39].

They not only experienced an increase in popularity but also seem to be a major suspect for the decline of other classical communication means, such as mailing lists and forums. We start by analyzing the platforms actually used by projects in our dataset. The scraping, based on regular expressions (see Section 2.4), took place between Aug 28 2022 at 21:01 and Aug 30 2022 at 02:12, leading to 12,081 scraped projects. Of those, 6,924 (57.3%) mention at least one such modern communication platform in their README files: 2,897 had 1 platform link, while 4,027 had 2 or more platform links. The remaining 5,157 projects had no platform links. The percentage is higher than the one reported by Käfer et al. [31] (57.3% vs. 46.7%), which can be explained by the fact that their analysis dates back 4 years.

We grouped communication platforms into three main categories: asynchronous, instant messaging, and social media.

Figure 4 summarizes number of links in READMEs (*Links*) and number of projects with at least one link (*Projects*) for each type of platform.



■ **Figure 4** Platform types.

■ **Figure 5** Number of projects linking to at least one platform.

There are Instant Messaging platforms (e.g., IRC, Slack, Discord), where communication can happen in real-time. In Asynchronous platforms (e.g., forum, mailing list, GitHub issues or discussions), communication usually takes some time to be processed and made available to other community members. The boundary between the two types has been blurred in the recent years by the technological improvements to the supporting infrastructure. We also considered Social Media platforms (e.g., Facebook, Twitter, Youtube). While their technical features can overlap with the other types, their huge user-base and ease of forming social connections make them stand out. Table 3 shows a complete list of the platforms we considered with a short description.

A project README can have multiple links to a single platform. This is particularly true for Social Media links where Twitter accounts of the main contributors or maintainers are all referenced.

■ **Table 3** Communication Platforms.

Platform	Description
Discord [16]	Voice, video, text messaging multimedia platform
Facebook [37]	Social media and social networking service
Forum	General category for web based discussion sites
GitHub [24]	GitHub infrastructure for project development
Gitter [39]	Voice, video, text messaging multimedia platform
IRC	Text-based instant messaging chat system
Linkedin [35]	Business social media & professional networking
Mailing List	E-mail based communication among recipients
Matrix [70]	Communication protocol implemented by clients
Medium [1]	Online publishing platform and social journalism
Reddit [52]	Social news aggregation, rating, discussion, and multimedia sharing
Slack [60]	Voice, video, text messaging multimedia platform
Spectrum [36]	Text-based web instant messaging chat system
StackOverflow [64]	Question and Answer website
Telegram [69]	Voice, video, text messaging multimedia platform
Twitter [75]	Social media and social networking service
Youtube [25]	Video hosting and sharing platform

In Figure 4, we see that the number of projects that use a specific platform is significantly lower than the number of links. For example, project *OpenAPITools/openapi-generator* [42] mentions 20 different Twitter accounts and 17 YouTube resources.

The identified categories are only a rough means to group similar platforms. In Figure 5 we show the number of projects having at least one reference to a specific platform.

Given our initial input set, it is not surprising to find GitHub to be the most referenced: The infrastructure is integrated enough to warrant support for the community with its own Issues and Discussions systems. This uniform consensus is followed by a more fragmented mix of Twitter, Discord, Gitter, Forums, Mailing Lists, and others in decreasing order of “popularity.” Far from being irrelevant, these platforms are used by hundreds of projects exclusively or in synergy. The next section sheds light on these synergies, complementarities, and on the competition between similar platforms.

5 Coexistence and Competition

Communication platforms can have different features and cater to different audiences. To cover development or users’ needs, projects can opt for using multiple media at the same time. What choices are made by core developers in terms of number and variety of platforms to include in a README?

Figure 6 depicts a non-exhaustive list of examples of overlaps between communication platforms (extracted with the REs presented in Section 2.4) used exclusively and side-by-side.

Around 38% of projects that use either Discord or Slack also include GitHub Issues in their READMEs (Figure 6a).

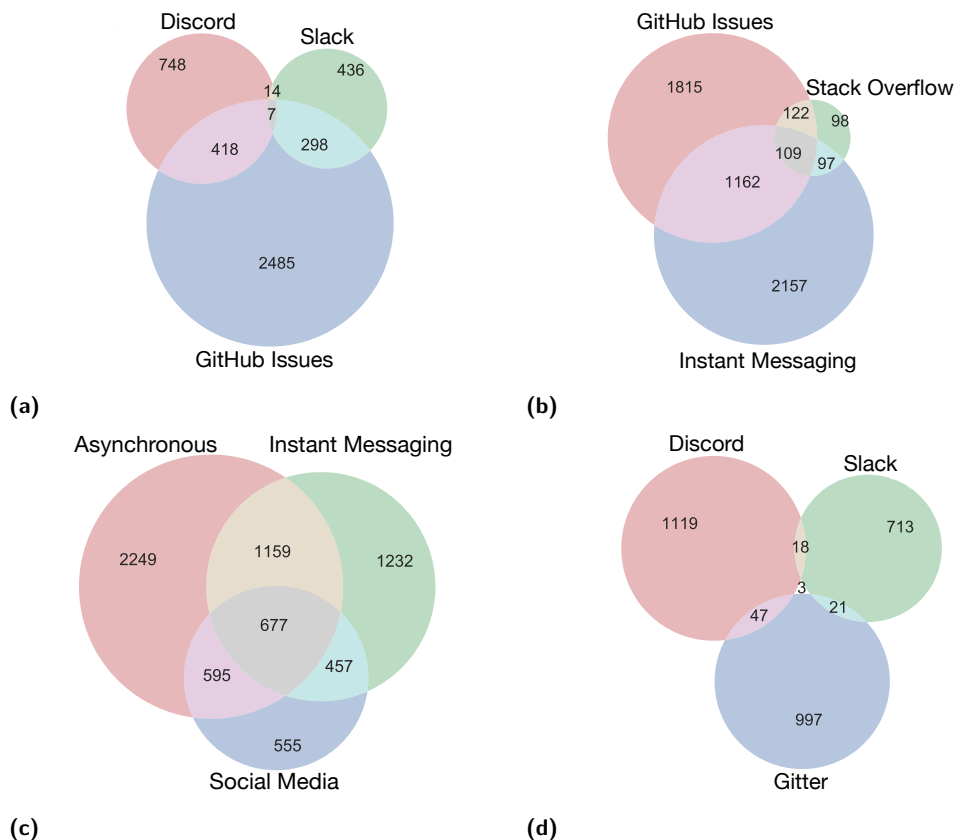


GitHub Issues can also be used without an explicit link in the README, as just a tab of the project, if enabled. Some platforms may be implicitly assumed to be available even if not present in the README.

Overall, 2,105 out of 3,208 projects (66%) using GitHub Issues, also have other communication platforms referenced in the README. Similar ratios are found, for example, for Discord with 801 out of 1,187 projects (67%).



Multiple communication platforms of different types can and do coexist.



■ **Figure 6** Communication Platforms Overlaps.

GitHub has significant overlaps with the whole category of instant messaging, and with specific asynchronous platforms (e.g., StackOverflow, see Figure 6b). However, 1,270 projects rely only on the integrated support provided by GitHub.

In general, if we consider the three main categories, we find that asynchronous platforms are used exclusively in 48% of projects, instant messaging follows with 35%, and social platforms seem the most frequently used as a complementary option (76%, see Figure 6c).

⚠️ | *It is not clear if different categories are mutually exclusive and why in a considerable amount of projects they tend to be used in conjunction.*

💡 | *This analysis should be complemented by how the user-base is distributed over these platforms.*

6 Instant Messaging: A Deep Dive

What makes instant messaging platforms appealing to developers? The steady growth in the number of projects including at least one platform of this kind is a piece of evidence supporting the need for fast and rich communication.

Instant messaging platforms fulfill a very specific role: Providing communication in real-time, possibly with rich media sharing capabilities (e.g., links, videos, files), and Voice over IP conferencing (i.e., VoIP). Two instances of these platforms are seldom found together. Similar characteristics, audiences, and usages make competition the prevailing paradigm.

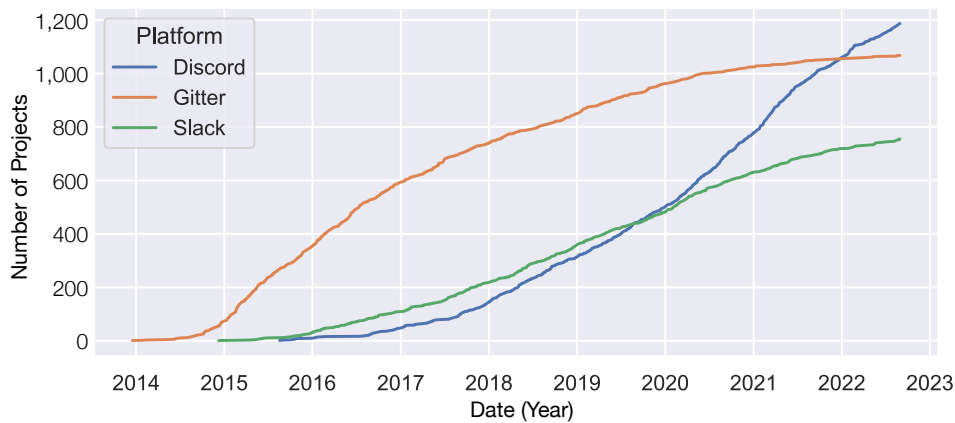
Figure 6d shows that 97% of projects opting for these platforms choose between one of the three alternatives. Three projects include links (see Section 2.4) to all the platforms and also other instant messaging (e.g., Spectrum), but only *PowerShell/PowerShell* has a significant Discord community (more than 10k members).



Gitter, Discord, and Slack are selected by projects as alternatives, very seldom coexisting. This can be a possible strategy for successful projects not to spread their community too thin over multiple platforms with similar capabilities.

6.1 Gitter, Discord, and Slack: A Timeline

Based on README history and mining links for each version of the README as detailed in Section 2.5, for each project, we look for the first appearance date of Gitter, Discord, and Slack (Figure 7).



■ **Figure 7** Timeline of cumulative adoption date of Slack, Discord, and Gitter.

Gitter appeared for the first time at the end of 2013, followed one year later by Slack, and then Discord after 8 months. All three platforms show a “ramp-up” period of slightly more than one year after their first appearance, followed by a steady growth at different rates. Both Gitter (in mid-2020) and Slack (in 2022) reached a *plateau* where just a handful of projects added them to their communication platforms in the last year. Discord, on the other hand, is still growing significantly.

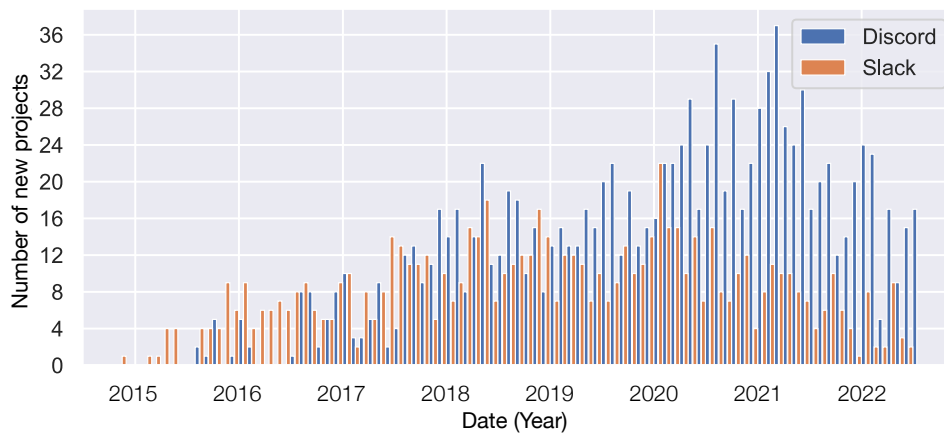
Since the beginning of 2020, Discord consistently outperformed Slack in terms of number of new projects adopting the platform for their community (Figure 8). The monthly growth rate has been higher than the highest for Slack in the previous years. It has also been at higher levels more consistently and for a longer period.

The comparison between additions of Gitter and Discord (Figure 9) shows a similar or even more evident tendency. The decline of the former and the growth of the latter are almost perfectly mirroring each other.

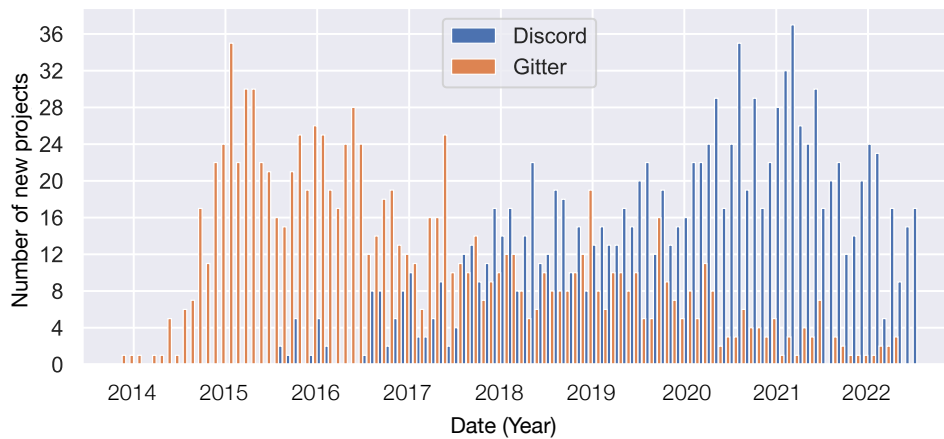


While one platform stops being added to projects, another is on the rise. This happened in the past and is bound to happen again in the future.

There is no guarantee that the example of the Spectrum platform we highlighted in Section 1 will be followed when Gitter goes out of fashion. It is also possible that the entire history of discussions, bug fixing sessions, and design decisions will just disappear.



■ **Figure 8** Monthly new projects adopting Discord and Slack.



■ **Figure 9** Monthly new projects adopting Gitter and Discord.

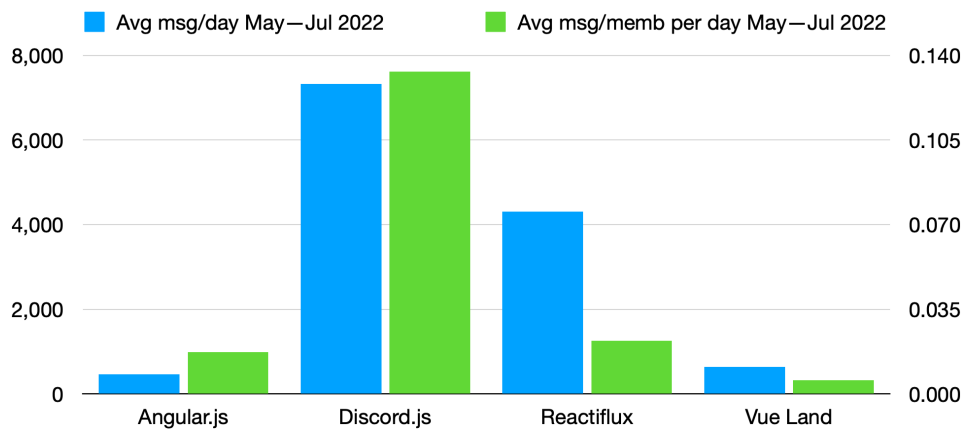
6.2 Throughput and Volatility

We investigated four Discord communities. Reactflux, Vue Land, and Angular.js are respectively tied to React, Vue, and Angular (web development frameworks). We compared them with each other and with the Discord.js community (Discord bot development).

In Figure 10, we show how the average number of messages per member in a sample period of three months (i.e., May–July, 2022) has high variability. While this might be due to a number of factors, we are interested in the sheer scale of the messages exchanged on those platforms every day.

Around 550 messages are exchanged per day in Vue Land and Angular.js. In Discord.js, instead, users exchange 305 messages *each hour*, totaling more than 7k messages a day.

The throughput of these servers means information is lost if one does not pay attention to notifications. Only a few messages are visible at a time and they scroll up quickly, putting full conversations behind the event horizon in a matter of minutes. Alert filters and community policies (e.g., forbidden mentioning of server wide tags) can only partially mitigate this problem. The trade-off between losing potentially interesting discussions and being constantly interrupted by notifications is the choice many modern developers face when dealing with these kinds of communities.



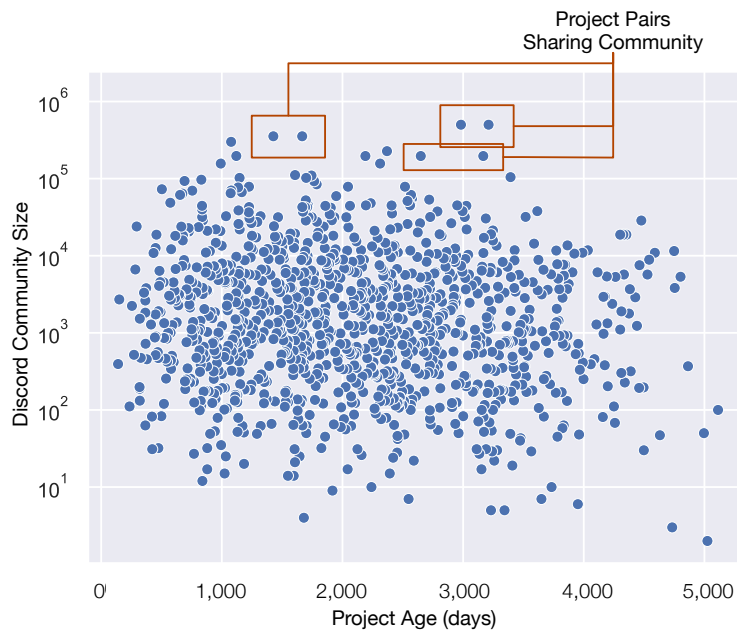
■ **Figure 10** Messages per day and average messages per day per member from May to July 2022 for four example Discord servers.



Application of summarization, visualization, and information retrieval techniques is fundamental to deal with scalability problems of these platforms.

6.3 Community Sizes

Discord communities in our dataset vary in size between 2 and 500,000 members. Figure 11 depicts Discord community size with respect to project age (i.e., days since creation).



■ **Figure 11** Discord community size with respect to project age (days from creation).



This should be investigated more in-depth to see if it is a breakpoint at which particular actions should be taken to keep the community growing.



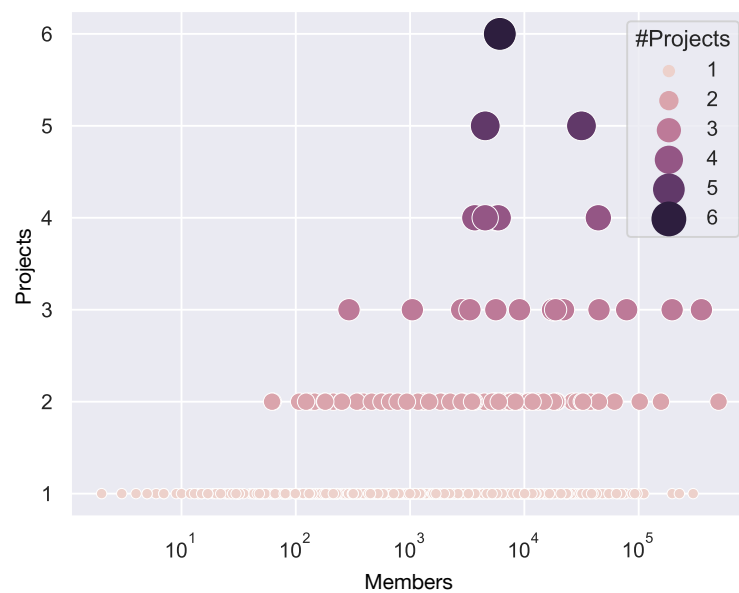
Extraction of Slack community sizes has proven more difficult due to the high variance in invite page formats. Gitter does not even have an invite page to scrape, and, to the best of our knowledge, the community size cannot be automatically retrieved.

6.4 Different Projects, Same Community

Being in the same Discord community means sharing the same server (i.e., each project has a link in the README, possibly with different formats, but pointing to the same Discord server). We consider this a case of “different projects, same community”.

Figure 11 shows horizontal pairs in the top part of the scatterplot, suggesting that different projects might share the same community: Our initial hypothesis that “same size of the community means same community” might not apply, especially for smaller communities. Nevertheless, it is unlikely for two different large communities to have the same number of members at the same time. We manually inspected the projects in those pairs and they are indeed different projects referring to the same wider community. For Discord we could reliably use the community name to confirm our hypothesis, as parsed from the invitation metadata (Section 2.4).

Figure 12 shows how many projects share a community with respect to community size.



■ **Figure 12** Projects referencing the same community.



Further analysis can show if projects are tightly coupled (e.g., different projects from the same organization, new major versions of the same project) or if different projects have an underlying reason to cater to the same audience.

6.5 Technical, Social, and Ethical Challenges

Some community platforms are public, some allow anonymous access, some require a form of registration or access permission. We found communities with (automatic) procedures to accept new members but, in general, it is hard to devise a general automatic “agent” to explore all of them.

The sheer amount of customization that is possible, even in a simple invite landing page of Slack, has been an obstacle to getting reliable data about communities lying behind those pages. Exploring a larger and more varied sample could shed light on platform dependent similarities and differences.



Machine readable APIs for communities can greatly benefit not only research in this field, but also open new possibilities, e.g., the automatic migration of community-generated content to preserve the history of projects when the underlying technologies evolve.

Socio-ethical challenges. Communities usually are digital aggregations of people’s thoughts, ideas, rants, strengths, and weaknesses. Collecting such information can be seen as poking inside someone’s house. One can do it if having legitimate reasons to do so. One can be welcome if providing benefits for the community. But one can also be faced with concerns about privacy and legitimate use of collected information.

Companies owning the platform sometimes are more keen to share their data than administrators of communities they host. While Slack has a monetization policy tied to its history retrievability, Discord allows unlimited access to a wealth of historical information via its API.

A big role in the extensibility of our study is played by the attitude of administrators of interesting communities. While information might be public (i.e., anyone with a Discord account can automatically join a server and browse its entire content), to comply with Discord’s Terms of Service we need to ask for permission to add a bot to extract useful information for DWARVENMAIL. In this crucial step a fundamental role is played by personal beliefs and perceptions of the benefits of such a bot by the administrators and the community itself.



Accessibility of information is ever more beyond the fence of what is technically possible, towards the barrier of what is ethically and socially accepted.

7 Threats to Validity

Our analyses are based on a dataset of public GitHub projects as the only source. This poses a threat to the generalizability of our results with respect to the type of projects hosted on GitHub. Open source projects developed in this social coding style might differ significantly from closed source projects developed by a small team of hired developers. There are also no guarantees that the results presented can be generalized to projects hosted on other similar repositories (e.g., SourceForge).

The current study presents a limited generalizability with respect to the format of README files. Although our sampling procedure (Section 2.3) ensures generalizability with respect to the project’s main programming language, different README file formats could provide different link types and formats not fully captured by our analysis.



We found evidence of more than 15 different README formats. While most share a similar structure for external links, systematic analysis of these formats could improve the generalizability of the results.

Limiting the extraction of the documentation landscape to what is reachable from the main README file (i.e., ignoring links to other READMEs in submodules of a project) poses a threat to construct validity. This threat is partially mitigated by the magnitude of the phenomenon we highlighted, emerging despite the limited scope, and calling for discussion and further investigation (i.e., also considering auxiliary documentation sources as a starting point to map the landscape).

43:18 On the Rise of Modern Software Documentation

Our analysis benefits from verifying the validity of links whenever possible (i.e., if the resource referred by the link is still available we expect an HTTP 200 OK response). When mining GitHub we verified the links we found in a two step process. The time interval between the first pass for scraping and the second pass for verification was short enough to guarantee that most links were in their intended state. Obsolete links may be possible and are part of the present study.



The analysis lacks accuracy when links are redirected or reused. Moreover, in the effort to reconstruct link patterns for previous standard link formats of some platforms (e.g., Slack) we adopt a conservative approach where if the format follows reasonable patterns it is accepted as a valid link in the history of a README file. We have no guarantee nor a way to discover if the link was valid in the past.

The only possibility to study the evolution and validity of such links is to constantly monitor README files and their evolution over a period of time. Link validity can be checked as soon as the change in the README is triggered. This kind of study is outside of the scope of the presented work.



Semantic analysis of the pointed links could improve relatedness, reducing false positives in link validity. Automatic link validity and relatedness to the source topic should be investigated.

Links that are not visually represented in the rendered README are currently part of the analyses. This threat to the validity of our conclusions is partially mitigated by the low frequency of such occurrences. We found only 3 non-rendered links in 2 manually annotated projects (0.1% of links, 3.3% of projects).



We did not perform an analysis based on project types. Relationships between project type, intended audience, and the resulting documentation landscape could provide insights on how to leverage the landscape for projects of different natures and at different maturity stages.

8 Related Work

Communication channels, especially those tightly coupled with collaborative development platforms (e.g., GitHub), are fundamental for successful software development.

Hoegl et al. [29] and Lindsjrn et al. [34] found communication to be an essential subcontract of teamwork quality. Tantisuwankul et al. analyzed the communication channels of GitHub projects [68]. Studying 70k library projects in 7 ecosystems, they identified 13 communication channels as “a form of knowledge transfer or sharing” (e.g., licenses, change logs). They found that GitHub projects adopt multiple channels, which change over time, to capture new and update existing knowledge. Storey et al. [65] conducted a large-scale survey with 1,449 GitHub users to understand the communication channels developers find essential to their work. On average, developers indicated they use 11.7 channels across all their activities (e.g., email, chat, microblogging, Q&A websites). They concluded that “communication channels shape and challenge the participatory culture in software development.” Hata et al. [27] studied early adopters of GitHub Discussions, finding that developers considered them useful and important. Lima et al. [32] used NLP to detect related discussions of OSS communities in GitHub Discussions.

Treude and Storey [74] interviewed users of a community portal, finding that clients, developers, and end-users are involved in the process of externalizing developer knowledge.

Nugroho et al. [40] studied how Eclipse developers utilize project forums, concluding that forums are essential platforms for linking various resources in the Eclipse ecosystem besides representing an important source of expert knowledge.

Modern social media are becoming another information source for development activities. Mezouar et al. [38] studied how tweets can improve the bug fixing process. They observed how issues for Firefox and Chrome are usually reported earlier through Twitter than on tracking systems. This can potentially decrease the lifespan of a bug. Guzman et al. [26] analyzed the usage characteristics, content, and automatic classification of tweets about software applications. They found that tweets contain useful information for software companies but stressed the need for automatic filtering of irrelevant information.

Instant messaging platforms, from Internet Relay Chat (IRC) to Discord, went from simple text messages to rich multimedia support with integrated DevOps workflows (i.e., Slack integrations). Yu et al. [78] learned how real-time (i.e., IRC) and asynchronous (i.e., mailing lists) communications were used and balanced across the GNOME GTK+ project. Shihab et al. [59] analyzed IRC meeting logs and found that developers actively contributed through meeting channels.

Lin et al. [33] argued Slack played an increasingly significant role, sometimes replacing emails. They found various benefits of Slack over mailing lists. Developers use it for team-wide purposes (e.g., communicating with teammates, file and code sharing), community support (e.g., special interest groups), and personal benefits (e.g., networking, social activities). They also observed that developers commonly used bots to support their work. Chatterjee et al. [9] analyzed the conversations of developers from five Slack programming communities and developers' StackOverflow posts. They found prevalent useful information, including API mentions and code snippets with descriptions in both sources.

Alkadhi et al. [5] examined “rationale” elements (i.e., discussed issues, alternatives, pro-/con-arguments, decisions) in Atlassian HipChat messages of three software development teams. They found frequent, valuable discussions with elements of rationale. However, they also emphasized the need for automated tools due to the high volume of chat messages.

Shi et al. [58] conducted an empirical study on developers' Gitter chats. They manually analyzed 749 dialogs and performed an automated analysis of over 173K dialogs of OSS communities. Interestingly, developers tend to converse more on Wednesdays and Thursdays. They also found interaction patterns among conversations and noticed that developers tend to discuss topics such as API usage and errors. They argue the need for better utilization and mining of knowledge embedded in the massive chat history of OSS communities.

Hata et al. analyzed links in source code comments [28] in a large-scale study (~10 million links) extracted from files of the main language of the project. We focus on README file links, which are independent of the project language.

Ebert et al. [17] conducted an empirical study to understand which communication channels are used in GitHub projects and how they are presented to the audience, finding that the most common were chats, mail-related, social media, and GitHub channels. Käfer et al. [31] analyzed GitHub communication channels, finding that “Mailing lists are being replaced by modern enterprise chat systems in OSS development.” Our work broadens the scope beyond communication channels and adds details needed to identify the current status, understand how it has evolved, and obtain meaningful insights on why this is happening.

Each of the previously discussed studies focuses on a specific part of the documentation landscape, recognizing the importance of the sources for knowledge management and documentation. What is still missing is a higher level understanding of the phenomenon that shifts the relative importance of these sources over time, intra- and inter-project.

9 Conclusions and Future Work

Classical software documentation is being replaced by “communication”. At least in open source software on GitHub, it is supported by a plethora of platforms characterized by high throughput, volatility, and heterogeneity. The original vision of on-demand developer documentation [55] advocated for a paradigm shift. A shift did happen, but it was not in the direction foreseen by Robillard et al. five years ago. The new communication platforms bring new challenges and opportunities for modern software documentation. It is time to shed light on new forms of documentation. A comparison with classical documentation and where it survives, unscathed by the new media and the needs of modern software development, might help rethink the role of documentation itself. Research efforts in this direction can help maintain documentation useful for software comprehension, maintenance, and evolution, independently of the form it will take.

To achieve this we need a better understanding of the phenomenon occurring to software documentation sources. We regard the present work as scratching the surface of what has turned into an emerging heterogeneous, complex, and ever-changing documentation landscape, a *terra incognita* full of possibilities and threats.

References

- 1 A Medium Corporation. Medium. URL: <https://medium.com/>.
- 2 Tim Abbott. Why Slack’s free plan change is causing an exodus. URL: <https://blog.zulip.com/2022/08/26/why-slacks-free-plan-change-is-causing-an-exodus/>.
- 3 Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: The practitioners’ perspective. In *Proceedings of ICSE 2020 (International Conference on Software Engineering)*, pages 590–601. ACM, 2020.
- 4 Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *Proceedings of ICSE 2019 (International Conference on Software Engineering)*, pages 1199–1210. IEEE/ACM, 2019.
- 5 Rana Alkadhi, Teodora Lata, Emitza Guzman, and Bernd Bruegge. Rationale in development chat messages: An exploratory study. In *Proceedings of MSR 2017 (International Conference on Mining Software Repositories)*, pages 436–446. IEEE/ACM, 2017.
- 6 Maurício Aniche, Christoph Treude, Igor Steinmacher, Igor Wiese, Gustavo Pinto, Margaret-Anne Storey, and Marco Aurélio Gerosa. How modern news aggregators help development communities shape and share knowledge. In *Proceedings of ICSE 2018 (International Conference on Software Engineering)*, pages 499–510. ACM, 2018.
- 7 Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of ICSE 2013 (International Conference on Software Engineering)*, pages 712–721. IEEE, 2013.
- 8 Hudson Borges and Marco Tulio Valente. What’s in a GitHub star? Understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.
- 9 Preetha Chatterjee, Kostadin Damevski, Lori Pollock, Vinay Augustine, and Nicholas A Kraft. Exploratory study of Slack Q&A chats as a mining source for software engineering tools. In *Proceedings of MSR 2019 (International Conference on Mining Software Repositories)*, pages 490–501. IEEE/ACM, 2019.
- 10 Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, 82(6):981–992, 2009.

- 11 Codecov. Codecov. URL: <https://about.codecov.io/>.
- 12 David Curry. Slack revenue and usage statistics (2022). URL: <https://www.businessofapps.com/data/slack-statistics/>.
- 13 Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in GitHub for MSR studies. In *Proceedings of MSR 2021 (International Conference on Mining Software Repositories)*, pages 560–564. IEEE/ACM, 2021.
- 14 Barthélémy Dagenais and Martin P Robillard. Creating and evolving developer documentation: Understanding the decisions of open source contributors. In *Proceedings of FSE 2010 (International Symposium on Foundations of Software Engineering)*, pages 127–136. ACM, 2010.
- 15 Discord. Invites 101. URL: <https://support.discord.com/hc/en-us/articles/208866998-Invites-101>.
- 16 Discord, Inc. Discord. URL: <https://discord.com/>.
- 17 Verena Ebert, Daniel Graziotin, and Stefan Wagner. How are communication channels on GitHub presented to their intended audience? – A thematic analysis. In *Proceedings of EASE 2022 (International Conference on Evaluation and Assessment in Software Engineering)*, pages 40–49. ACM, 2022.
- 18 Osama Ehsan, Safwat Hassan, Mariam El Mezouar, and Ying Zou. An empirical study of developer discussions in the Gitter platform. *Transactions on Software Engineering and Methodology*, 30(1):1–39, 2020.
- 19 Andrew Forward and Timothy C Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of DocEng 2002 (Symposium on Document Engineering)*, pages 26–33. ACM, 2002.
- 20 freeCodeCamp. Our experience with Slack. URL: <https://www.freecodecamp.org/news/so-yeah-we-tried-slack-and-we-deeply-regretted-it-391bcc714c81>.
- 21 Golar Garousi, Vahid Garousi-Yusifoglu, Guenther Ruhe, Junji Zhi, Mahmoud Moussavi, and Brian Smith. Usage and usefulness of technical software documentation: An industrial case study. *Information and Software Technology*, 57:664–682, 2015.
- 22 GitHub. Fork a repo. URL: <https://docs.github.com/en/get-started/quickstart/fork-a-repo>.
- 23 GitHub. PyGithub. URL: <https://github.com/PyGithub/PyGithub>.
- 24 GitHub, Inc. GitHub. URL: <https://github.com/>.
- 25 Google, LLC. YouTube. URL: <https://www.youtube.com/>.
- 26 Emitza Guzman, Rana Alkadhi, and Norbert Seyff. A needle in a haystack: What do Twitter users say about software? In *Proceedings of RE 2016 (International Requirements Engineering Conference)*, pages 96–105. IEEE, 2016.
- 27 Hideaki Hata, Nicole Novielli, Sebastian Baltes, Raula Gaikovina Kula, and Christoph Treude. GitHub Discussions: An exploratory study of early adoption. *Empirical Software Engineering*, 27(1):1–32, 2022.
- 28 Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. 9.6 million links in source code comments: Purpose, evolution, and decay. In *Proceedings of ICSE 2019 (International Conference on Software Engineering)*, pages 1211–1221. IEEE, 2019.
- 29 Martin Hoegl and Hans Gemuenden. Teamwork quality and the success of innovative projects: A theoretical concept and empirical evidence. *Organization Science*, 12(4):435–449, 2001.
- 30 Jialun Aaron Jiang, Charles Kiene, Skyler Middler, Jed R. Brubaker, and Casey Fiesler. Moderation challenges in voice-based online communities on Discord. *Proceedings of HCI 2019 (Human-Computer Interaction)*, 3(CSCW):1–23, 2019.
- 31 Verena Käfer, Daniel Graziotin, Ivan Bogicevic, Stefan Wagner, and Jasmin Ramadani. Communication in Open-Source projects – End of the e-mail era? In *Proceedings of ICSE 2018 (International Conference on Software Engineering)*, pages 242–243. ACM, 2018.
- 32 Marcia Lima, Igor Steinmacher, Denae Ford, Evangeline Liu, Grace Vorreuter, Tayana Conte, and Bruno Gadelha. Looking for related discussions on GitHub Discussions. In *arXiv*, 2022.


- 33 Bin Lin, Alexey Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. Why developers are slacking off: Understanding how software teams use Slack. In *Proceedings of CSCW/SCC 2016*, pages 333–336. ACM, 2016.
- 34 Yngve Lindsjørn, Dag I.K. Sjøberg, Torgeir Dingsøy, Gunnar R. Bergersen, and Tore Dybå. Teamwork quality and project success in software development: A survey of agile development teams. *Journal of Systems and Software*, 122:274–286, 2016.
- 35 LinkedIn Corporation. LinkedIn. URL: <https://www.linkedin.com>.
- 36 Brian Lovin. Join us on our new journey. URL: https://web.archive.org/web/20220927203327/https://spectrum.chat/spectrum/general/join-us-on-our-new-journey_e4ca0386-f15c-4ba8-8184-21cf5fa39cf5.
- 37 Meta. Facebook. URL: <https://www.facebook.com/>.
- 38 Mariam El Mezouar, Feng Zhang, and Ying Zou. Are tweets useful in the bug fixing process? An empirical study on Firefox and Chrome. *Empirical Software Engineering*, 23(3):1704–1742, 2018.
- 39 New Vector, Ltd. Gitter. URL: <https://gitter.im/>.
- 40 Yusuf Sulisty Nugroho, Syful Islam, Keitaro Nakasai, Ifraz Rehman, Hideaki Hata, Raula Gaikovina Kula, Meiyappan Nagappan, and Kenichi Matsumoto. How are project-specific forums utilized? A study of participation, content, and sentiment in the Eclipse ecosystem. *Empirical Software Engineering*, 26(6):132, 2021.
- 41 N. Nurmaliani, D. Zowghi, and S. P. Williams. Using card sorting technique to classify requirements change. In *Proceedings of IREC 2004 (International Requirements Engineering Conference)*, pages 240–248. IEEE, 2004.
- 42 OpenAPI Tools. OpenAPI Generator. URL: <https://github.com/OpenAPITools/openapi-generator>.
- 43 Dennis Pagano and Walid Maalej. How do developers blog? An exploratory study. In *Proceedings of MSR 2011 (Working Conference on Mining Software Repositories)*, pages 123–132. ACM, 2011.
- 44 Papyrus. Easy company intranet & internal team wiki for Slack. URL: <https://papyrus.com/slack-wiki-intranet/>.
- 45 Esteban Parra, Mohammad Alahmadi, Ashley Ellis, and Sonia Haiduc. A comparative study and analysis of developer communications on Slack and Gitter. *Empirical Software Engineering*, 27(2):1–33, 2022.
- 46 Esteban Parra, Ashley Ellis, and Sonia Haiduc. GitterCom: A dataset of Open Source developer communications in Gitter. In *Proceedings of MSR 2020 (International Conference on Mining Software Repositories)*, pages 563–567. ACM, 2020.
- 47 Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of MSR 2014 (Working Conference on Mining Software Repositories)*, pages 102–111. IEEE/ACM, 2014.
- 48 Python Software Foundation. Python Package Index. URL: <https://pypi.org/>.
- 49 Marco Raglianti, Roberto Minelli, Csaba Nagy, and Michele Lanza. Visualizing Discord servers. In *Proceedings of VISSOFT 2021 (Working Conference on Software Visualization)*, pages 150–154. IEEE, 2021.
- 50 Marco Raglianti, Csaba Nagy, Roberto Minelli, and Michele Lanza. Using Discord conversations as program comprehension aid. In *Proceedings of ICPC 2022 (International Conference on Program Comprehension)*, pages 597–601. ACM, 2022.
- 51 Marco Raglianti, Csaba Nagy, Roberto Minelli, Bin Lin, and Michele Lanza. Replication package. URL: <https://figshare.com/s/33c8af534dba61d72c41>.
- 52 Reddit. Reddit. URL: <https://www.reddit.com/>.
- 53 Lionel P Robert and Alan R Dennis. Paradox of richness: A cognitive model of media choice. *IEEE Transactions on Professional Communication*, 48(1):10–21, 2005.

- 54 Martin P Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.
- 55 Martin P. Robillard, Andrian Marcus, Christoph Treude, Gabriele Bavota, Oscar Chaparro, Neil Ernst, Marco Aurélio Gerosa, Michael Godfrey, Michele Lanza, Mario Linares-Vásquez, Gail C. Murphy, Laura Moreno, David Shepherd, and Edmund Wong. On-demand developer documentation. In *Proceedings of ICSME 2017 (International Conference on Software Maintenance and Evolution)*, pages 479–483. IEEE, 2017.
- 56 Hareem Sahar, Abram Hindle, and Cor-Paul Bezemer. How are issue reports discussed in Gitter chat rooms? *Journal of Systems and Software*, 172:110852, 2021.
- 57 Benjamin Saunders, Julius Sim, Tom Kingstone, Shula Baker, Jackie Waterfield, Bernadette Bartlam, Heather Burroughs, and Clare Jinks. Saturation in qualitative research: Exploring its conceptualization and operationalization. *Quality & Quantity*, 52(4):1893–1907, 2018.
- 58 Lin Shi, Xiao Chen, Ye Yang, Hanzhi Jiang, Ziyong Jiang, Nan Niu, and Qing Wang. A first look at developers’ live chat on Gitter. In *Proceedings of ESEC/FSE 2021 (European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*, pages 391–403. ACM, 2021.
- 59 Emad Shihab, Zhen Ming Jiang, and Ahmed E Hassan. On the use of internet relay chat (IRC) meetings by developers of the GNOME GTK+ project. In *Proceedings of MSR 2009 (Working Conference on Mining Software Repositories)*, pages 107–110. IEEE, 2009.
- 60 Slack Technologies. Slack. URL: <https://slack.com/>.
- 61 Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015.
- 62 Sonatype. Maven Central Repository. URL: <https://central.sonatype.dev/>.
- 63 Donna Spencer. *Card Sorting: Designing Usable Categories*. Rosenfeld Media, 2009.
- 64 Stack Exchange, Inc. Stack Overflow. URL: <https://stackoverflow.com/>.
- 65 Margaret-Anne Storey, Alexey Zagalsky, Fernando Figueira Filho, Leif Singer, and Daniel M. German. How social and communication channels shape and challenge a participatory culture in software development. *IEEE Transactions on Software Engineering*, 43(2):185–204, 2017.
- 66 Viktoria Stray and Nils Brede Moe. Understanding coordination in global software engineering: A mixed-methods study on the use of meetings and Slack. *Journal of Systems and Software*, 170:110717, 2020.
- 67 Keerthana Muthu Subash, Lakshmi Prasanna Kumar, Sri Lakshmi Vadlamani, Preetha Chatterjee, and Olga Baysal. DISCO: A dataset of Discord chat conversations for software engineering research. In *Proceedings of MSR 2022 (International Conference on Mining Software Repositories)*, pages 227–231. IEEE/ACM, 2022.
- 68 Jirateep Tantisuwankul, Yusuf Sulisty Nugroho, Raula Gaikovina Kula, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. A topological analysis of communication channels for knowledge sharing in contemporary GitHub projects. *Journal of Systems and Software*, 158:110416, 2019.
- 69 Telegram. Telegram. URL: <https://telegram.org/>.
- 70 The Matrix.org Foundation C.I.C. Matrix. URL: <https://matrix.org/>.
- 71 Mike Thelwall and Liwen Vaughan. A fair history of the web? Examining country balance in the Internet Archive. *Library & Information Science Research*, 26(2):162–176, 2004.
- 72 Yuan Tian, Palakorn Achananuparp, Ibrahim Nelman Lubis, David Lo, and Ee-Peng Lim. What does software engineering community microblog about? In *Proceedings of MSR 2012 (Working Conference on Mining Software Repositories)*, pages 247–250. IEEE, 2012.
- 73 Travis CI. Travis CI. URL: <https://www.travis-ci.com/>.
- 74 Christoph Treude and Margaret-Anne Storey. Effective communication of software development knowledge through community portals. In *Proceedings of ESEC/FSE 2011 (European Software Engineering Conference and Symposium on the Foundations of Software Engineering)*, pages 91–101. ACM, 2011.
- 75 Twitter, Inc. Twitter. URL: <https://twitter.com/>.

43:24 On the Rise of Modern Software Documentation

- 76 Jed R Wood and Larry E Wood. Card sorting: Current practices and beyond. *Journal of Usability Studies*, 4(1):1–6, 2008.
- 77 Zhou Yang, Chenyu Wang, Jieke Shi, Thong Hoang, Pavneet Kochhar, Qinghua Lu, Zhenchang Xing, and David Lo. What do users ask in open-source AI repositories? An empirical study of GitHub issues. *arXiv preprint arXiv:2303.09795*, 2023.
- 78 Liguu Yu, Srinivas Ramaswamy, Alok Mishra, and Deepti Mishra. Communications in global software development: An empirical study using GTK+ OSS repository. In *Proceedings of OTM 2011 (On the Move to Meaningful Internet Systems)*, pages 218–227. Springer, 2011.
- 79 Junji Zhi, Vahid Garousi-Yusifoglu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015.
- 80 Carlos Zimmerle, Kiev Gama, Fernando Castor, and José Murilo Mota Filho. Mining the usage of reactive programming APIs: A study on GitHub and Stack Overflow. In *Proceedings of MSR 2022 (International Conference on Mining Software Repositories)*, pages 203–214. ACM, 2022.
- 81 Zyte. Scrapy. URL: <https://scrapy.org>.

Python Type Hints Are Turing Complete

Ori Roth   

Department of Computer Science, Technion, Haifa, Israel

Abstract

Grigore proved that Java generics are Turing complete by describing a reduction from Turing machines to Java subtyping. Furthermore, he demonstrated that his “subtyping machines” could have metaprogramming applications if not for their extremely high compilation times. The current work reexamines Grigore’s study in the context of another prominent programming language – Python. We show that the undecidable Java fragment used in Grigore’s construction is included in Python’s type system, making it Turing complete. In contrast to Java, Python type hints are checked by third-party static analyzers and run-time type checkers. The new undecidability result means that both kinds of type checkers cannot fully incorporate Python’s type system *and* guarantee termination. The paper includes a survey of infinite subtyping cycles in various type checkers and type reification in different Python distributions. In addition, we present an alternative reduction in which the Turing machines are simulated in real time, resulting in a significantly faster compilation. Our work is accompanied by a Python implementation of both reductions that compiles Turing machines into Python subtyping machines.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases nominal Subtyping with Variance, Python

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.44

Category Pearl/Brave New Idea

Related Version *Previous Version*: <https://doi.org/10.48550/arxiv.2208.14755>

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*:
<https://doi.org/10.4230/DARTS.9.2.1>

Acknowledgements The author would like to thank to Yossi Gil for inspiring them to study the topic of nominal subtyping with variance.

1 Introduction

Python enhancement proposal (PEP) 484 introduced optional type hints to the Python programming language, together with a full-blown gradual type system [16]. Tools such as Mypy [9] use type hints to type-check Python programs. Certain programs, however, cause Mypy to enter an infinite loop (we show an example below). We argue that the reason behind these failures is not a Mypy bug, but a deeper issue in the PEP 484 type system. We use Grigore’s reduction from Turing machines (TMs) to nominal subtyping with variance [6] to prove that Python type hints are, in fact, Turing complete. In other words, checking whether a Python program is correctly typed is as hard as the halting problem.

1.1 Nominal Subtyping With Variance

Subtyping is a type system decision problem. Given types t and s , the type system should decide whether type t is a subtype of s , $t <: s$, meaning that every t object is also a member of s . For example, every string is an object, `str <: object`, but not every object is a string, `object <: str`. Subtyping is needed, for example, for checking variable assignments:



© Ori Roth;

licensed under Creative Commons License CC-BY 4.0

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 44; pp. 44:1–44:15



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



44:2 Python Type Hints Are Turing Complete

```
x: t = ...
y: s = x # t<:s
```

The second assignment compiles if and only if $t <: s$.

Object-oriented languages such as Scala, C#, and Python use *nominal subtyping with variance* [7]. *Nominal* means that subtyping is guided by inheritance. Type t is a subtype of type s if class t is a descendant of s :

```
class s: ...
class t(s): ...
x: t = ...
y: s = x # ✓
```

Variance enables variability in type arguments. By default, type $t[u]$ is a subtype of $t[v]$ if and only if $u = v$. If t 's type parameter is covariant, u can be any subtype of v , $u <: v$; if it is contravariant, then $u >: v$. In Python, variance is specified in an argument to `TypeVar`, the constructor of type parameters. For example, the following Python program uses contravariance and is correctly typed:

```
x = TypeVar("x", contravariant=True)
class t(Generic[x]): ... # t has a contravariant type parameter x
x: t[str] = t[object]() # ✓ (str<:object)
```

Using a reduction from the Post correspondence problem (PCP) [13], Kennedy and Pierce showed that nominal subtyping with variance is undecidable [7]. Their work focused on three subtyping features (characteristics):

1. *Contravariance*: The presence of contravariant type parameters, as described above.
2. *Expansive-recursive inheritance*: The closure of types under the inheritance relation and type decomposition ($t[v] \rightarrow v$) is unbounded. Intuitively, expansive inheritance requires class t to recur in one of its supertypes:

```
class t(Generic[x], s[s["t[t[x]]"]]): ...
# note: forward references are put in string literals
```

A formal definition of expansive-recursive inheritance is provided in Section 2.

3. *Multiple instantiation inheritance*: Class t is allowed to derive class $s[·]$ multiple times using different type arguments:

```
class t(s[object], s[str]): ... # not legal in Python
```

Kennedy and Pierce proved that subtyping becomes decidable when contravariance or expansive inheritance are removed, but they were uncertain about the contribution of multiple instantiation inheritance to undecidability.

1.2 Subtyping Machines

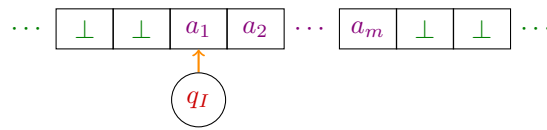
Ten years after them, Grigore showed that Java subtyping is Turing complete using a direct reduction from TMs [6]. This reduction uses a subset of Java that conforms to Kennedy and Pierce's nominal subtyping with variance. Grigore's result settled Kennedy and Pierce's open problem since Java does not support multiple instantiation inheritance [12, §8.1.5]. Intuitively, multiple instantiation inheritance corresponds to non-deterministic subtyping [7, 14], so while it was useful in the PCP reduction, it is redundant in the TM reduction because deterministic TMs are as expressive as non-deterministic TMs.

Grigore's reduction can be described as a function

$$e(M, w) = \Delta \cdot q \quad (1)$$

that encodes TM M and input word w as program $P = \Delta \cdot q$ consisting of class table Δ and subtyping query q . The encoding ensures that TM M accepts w if and only if query q type-checks against Δ .

The idea behind the reduction is to encode the TM configuration (instantaneous description) in the subtyping query q . Recall that the configuration of TM M comprises (i) the content of the memory tape, (ii) the location of the machine head on the tape, and (iii) the current state of the machine's finite control. Figure 1 illustrates the initial configuration of M : The input word $w = a_1 a_2 \dots a_m$ is written on the tape, the machine head points to the first letter a_1 , and the current state is the initial state q_I .



■ **Figure 1** The initial configuration of a Turing machine.

To explain how configurations are encoded as subtyping queries, let us first introduce some syntax (adopted from Grigore's paper). We write a generic type $\mathbf{A}[\mathbf{B}[\mathbf{C}]]$ as ABC for short. The use of \blacktriangleleft instead of $<$: in a subtyping query means that the type on the left-hand side should be read in reverse (the same goes for \blacktriangleright and $>$), e.g., $ABC \blacktriangleleft DE$ is equivalent to $CBA <: DE$.

The initial TM configuration, depicted in Figure 1, is encoded by the following subtyping query:

$$ZEE L_{\perp} N M^L N L_{a_1} N L_{a_2} N \dots N L_{a_m} N L_{\perp} Q_I^{wR} \blacktriangleleft EEZ \quad (2)$$

Observe that the types in Equation (2) have the same colors as the machine configuration elements in Figure 1 that they encode. For example, the type L_{a_1} encodes the tape symbol a_1 , and both are purple. As the type on the left-hand side is written in reverse, the content of the encoded tape can be obtained by reading the L types from the left to the right. The full legend is listed in Table 1.

■ **Table 1** The components of Grigore's subtyping machine. All types use a single contravariant type parameter x , except Z , which is monomorphic. The superscripts vary.

Type	Corresponding TM component / purpose
$L_{\perp} x$	an infinite blank tail of the tape
$L_{\sigma} x$	a tape cell containing the symbol σ
$M^L x$	the location of the machine head
$Q_I^{wR} x$	the current TM state q_I
Z, Ex, Nx	utility types without corresponding TM components

Grigore referred to the subtyping query in Equation (2) as a *subtyping machine* because when the subtyping algorithm tries to resolve it, it simulates the computation steps of the original TM. The subtyping deduction preserves the general structure of the query, except that it steadily pushes the state type Q_I along the tape. When Q_I reaches the head type M^L , the subtyping algorithm simulates a single TM transition: It overwrites the current tape

44:4 Python Type Hints Are Turing Complete

cell ($L_{a_1} \Rightarrow L_\sigma$), moves the machine head (by repositioning M^L), and changes the machine state ($Q_I \Rightarrow Q_J$). The resulting subtyping query correctly encodes the next configuration in the TM run. This process continues until the machine accepts and the query is resolved, or the machine aborts and a compilation error is raised. If the machine runs indefinitely, the subtyping algorithm does not terminate.

While TMs move the machine head to the left or right freely, subtyping machines can change direction only when reaching the end of the type, *EEZ*. After simulating a TM transition, the subtyping machine must reach the end(s) of the tape, rotate, and then reach the location of the machine head M in the right orientation, before it can simulate the next transition. In general, Grigore’s subtyping machines can make $O(m)$ operations for every computation step of the TM they simulate, where m is the number of symbols on the tape, resulting in a substantial slowdown. For example, Grigore’s simulation of the CYK algorithm, which usually runs in $O(n^3)$, takes $O(n^9)$ subtyping deduction steps to be completed.

1.3 Subtyping Metaprogramming

Beyond the undecidability result, Grigore demonstrated metaprogramming applications for their subtyping machines. Although the computational power of subtyping machines is unlimited, harnessing this power for conducting meaningful type-level metaprogramming as done in, e.g., C++ [18], is not easy. To integrate subtyping machines into programs, Grigore proposed to wrap them in fluent APIs [2]. Fluent API methods are called in a stream (chain) of consecutive invocations, as demonstrated in Listing 1.

■ **Listing 1** Running a subtyping machine with a fluent API.

```
p: Palindrome = a().b().b().a().b().b().a()
```

In Grigore’s design, the fluent chain produces the tape of the subtyping machine. For example, the chain in Listing 1 produces a types tape containing the input word *abbabba*. We run the subtyping machine by assigning the chain to a variable, invoking a the subtyping query in Equation (2).

The purpose of the subtyping machine is to validate a property of the chain at compile time. For example, a subtyping machine that recognizes palindromes forces the fluent chain to encode a palindrome word, or else it would not compile. Since the chain in Listing 1 encodes a palindrome, the subtyping machine accepts and the assignment type-checks. This sort of type-level metaprogramming in fluent APIs is employed for embedding domain-specific languages (DSLs) and enforcing higher-level API protocols [3, 10, 4, 19, 14]. In theory, Grigore’s subtyping machines can encode any computable DSL or API protocol. Unfortunately, the slowdown ingrained into the design of the subtyping machines results in extremely high compilation times, making the technique impractical.

1.4 Contributions

This work revisits Grigore’s study of subtyping machines in the context of Python. We show that Python’s type system includes the Java fragment used in Grigore’s construction and is, therefore, Turing complete. We review the impact of undecidable subtyping on the Python ecosystem, covering compile-time and run-time type checkers and different Python distributions. Finally, we present a new subtyping machine design that avoids the inherent slowdown imposed by Grigore’s construction – an essential step towards practical subtyping machine applications. Our subtyping machines simulate TMs in real time, i.e., make $O(1)$ operations for each TM transition. The paper is accompanied by a Python implementation of Grigore’s and our reductions, producing subtyping machines on top of Python type hints.

Outline

The rest of the paper is organized as follows. In Section 2, we show that Python type hints are Turing complete and discuss possible ways to make them more tractable. Section 3 includes the survey of infinite subtyping and type reification in Python. Section 4 introduces our alternative design for subtyping machines that simulate TMs in real time. Section 5 presents our Python implementations of Grigore’s original reduction and the new reduction, and compares their performances. Section 6 concludes.

2 Python Subtyping is Undecidable

Type hints were introduced into the Python programming language with PEP 484 [16]. PEP 484 defines the syntax of type hints but only provides an informal description of their semantics, referring the reader to the supplementary PEP 483 [17] for an in-depth discussion. Type hints are used as annotations and are entirely optional:

```
def positive(x: int) -> bool:
    return x > 0
```

Static analysis on type hints is not performed by the Python interpreter but by third-party tools. For instance, Mypy [9] is a type checker for Python type hints; in fact, PEP 484 was originally inspired by Mypy [16].

The type system described in PEP 484 supports declaration-site nominal subtyping with variance, similar to the abstract type system studied by Kennedy and Pierce [7]. Although originally designed for Java, Grigore’s subtyping machines conform to Kennedy and Pierce’s type system. To implement subtyping machines with Python type hints, we need to show that Python’s type system includes the two subtyping features essential for Grigore’s construction: contravariance and expansive-recursive inheritance.

In Python, type variables are specified using a special constructor, `TypeVar`. Making a contravariant (or covariant) type variable is as simple as passing an argument to `TypeVar`:

```
z = TypeVar("z", contravariant=True)
class N(Generic[z]): ... # class N has a contravariant parameter z
```

Expansive-recursive inheritance is a more elusive aspect of nominal subtyping. Kennedy and Pierce [7] defined expansive inheritance using the *inheritance and decomposition closure* $\text{cl}(t)$ of type t . Here we provide a brief description of the closure; the full definition can be found in Kennedy and Pierce’s paper. Recall that we use a shorthand notation for generic types, $Cs = \mathcal{C}[s]$. If $\text{cl}(t)$ contains the type Cs , then by decomposition it also contains s :

$$\text{(DECOMPOSITION)} \quad \frac{Cs \in \text{cl}(t)}{s \in \text{cl}(t)} \quad (3)$$

If, in addition, class $\mathcal{C}[x]$ inherits from type u , denoted $Cx : u$, then $\text{cl}(t)$ also contains the type $u[x \leftarrow s]$, in which every occurrence of type parameter x is substituted by s :

$$\text{(INHERITANCE)} \quad \frac{Cs \in \text{cl}(t) \quad Cx : u}{u[x \leftarrow s] \in \text{cl}(t)} \quad (4)$$

Kennedy and Pierce proved that a class table is expansive-recursive if and only if the set $\text{cl}(t)$ is infinite for some type t . For example, consider the following class declaration:

```
x = TypeVar("x")
class C(Generic[x], N[N["C[C[x]]"]]): ... # Cx:NNCCx
```

44:6 Python Type Hints Are Turing Complete

The inheritance of class C is expansive-recursive since the set $\text{cl}(Ct)$ is infinite for any type t :

1. $Ct \in \text{cl}(Ct)$
 2. $NNCCt \in \text{cl}(Ct)$ (INHERITANCE)
 3. $NCCT \in \text{cl}(Ct)$ (DECOMPOSITION)
 4. $CCT \in \text{cl}(Ct)$ (DECOMPOSITION)
 5. $NNCCCT \in \text{cl}(Ct)$ (INHERITANCE)
- and so on...
- (5)

Between steps 1 and 4 the type Ct was transformed to CCT , increasing the size of the type by a single C . By continuing the deduction we get that the set $\text{cl}(Ct)$ contains the type $C^n t$ for any $n \geq 1$ and is, therefore, infinite.

As PEP 484 and the accompanying PEP 483 do not mention any restrictions on expansive-recursive inheritance (at least, that the author could find), we conclude that Python implicitly supports expansive inheritance. As evidence, the example above correctly compiles in Python and Mypy.

By enabling both contravariance and expansive-recursive inheritance, the designers of PEP 484 opened up Python type hints to the same pitfalls of nominal subtyping with variance studied by Kennedy, Pierce, and Grigore. For example, the code in Listing 2, adapted from Kennedy and Pierce [7], shows how contravariance and expansive inheritance can be combined to induce an infinite subtyping cycle.

■ **Listing 2** Contravariance, expansive inheritance, and infinite subtyping with Python type hints.

```
from typing import TypeVar, Generic, Any
z = TypeVar("z", contravariant=True)
class N(Generic[z]): ...
x = TypeVar("x")
class C(Generic[x], N[N["C[C[x]]"]]): ...
class T: ...
class U: ...
_: N[C[U]] = C[T]() # CT <: NCU ✗ infinite subtyping
```

The last line of Listing 2 contains a variable assignment that invokes the subtyping query $CT <: NCU$. The query is resolved using two subtyping rules [7]: SUPER, allowing us to replace a type with its supertype using an inheritance rule:

$$\text{(SUPER)} \quad \frac{Cx : u \quad u[x \leftarrow s] <: t}{Cs <: t} \quad (6)$$

And VAR, allowing us to remove a single type from both sides of the query:

$$\text{(VAR)} \quad \frac{C\text{'s type parameter } x \text{ is contravariant} \quad t <: s}{Cs <: Ct} \quad (7)$$

We use the rules SUPER and VAR to resolve the query as follows:

1. $CT <: NCU$
 2. $NNCCT <: NCU$ (SUPER)
 3. $CU <: NCCT$ (VAR)
 4. $NNCCU <: NCCT$ (SUPER)
 5. $CCT <: NCCU$ (VAR)
- and so on...
- (8)

Between steps 1 and 5, the subtyping query was not reduced but increased in size. This deductive process continues indefinitely. When Mypy checks this program, it throws a segmentation fault. In Section 5, we show that Mypy crashes because it gets stuck in an infinite recursion during the subtyping algorithm.

Since contravariance and expansive inheritance are enough to implement Grigore’s subtyping machines, we get that Python type hints are Turing complete. Section 5 presents our implementation of Grigore’s reduction with Python type hints. The resulting subtyping machines correctly run with Mypy.

2.1 Taming Python’s Type System

The source of undecidability in PEP 484 is the hazardous combination of contravariance and expansive-recursive inheritance, whose presence enables the construction of Grigore’s subtyping machines. Kennedy and Pierce proved that nominal subtyping with variance becomes decidable once either contravariance or expansive inheritance are removed [7]. Restrictions to expansive inheritance are implemented, for example, in the Scala programming language [11, §5.1.5] and the .NET framework [1, §II.9.2]. Removing expansive inheritance, however, might not be enough to make Python type hints decidable. A more thorough inspection of the various features of Python’s type system is required to determine this.

Greenman et al. [5] and Tate et al. [15] presented alternative subtyping algorithms for Java that are decidable. Besides the theoretical work, they surveyed extensive corpora of Java projects to show that their new algorithms do not break existing code and are thus backward compatible. Future attempts at making Python’s type hints decidable may follow a similar line.

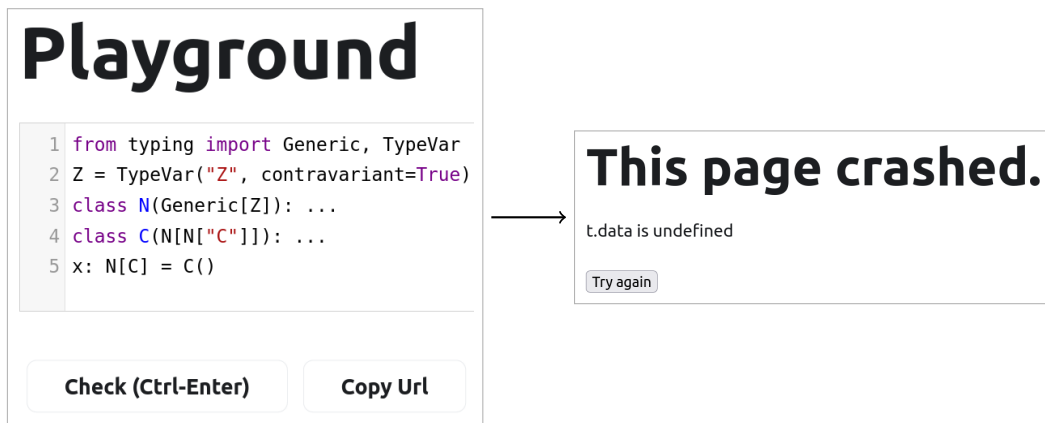
3 Type Reification and Infinite Subtyping in the Wild

A remote code execution vulnerability allows an attacker to run arbitrary *code* on a remote machine. Python type hints, on the other hand, can be used to run arbitrary *computations*. The distinction between code and computation is critical in the context of security: we can use type hints to compute prime numbers but not to access the file system or to open an SSH connection. Nevertheless, the fact that Python type hints are Turing complete does raise a few practical concerns. Type hints can induce infinite subtyping cycles, as demonstrated in Listing 2. These cycles cannot be detected by a “smart” type checker due to the undecidability of the halting problem. Thus, we can use infinite subtyping to attack Python type checkers and cause them to loop indefinitely or crash. For example, we managed to use infinite subtyping to crash the online playground of the Pyre Python type checker, as depicted in Figure 2. The playground is available at <https://pyre-check.org/play/>.

In Java, the compiler eliminates type arguments to generic types in a process called *type erasure*, making infinite subtyping a compile-time-only issue. Although PEP 484 mentions type erasure, it does not explicitly prohibit run-time type reification. The later PEP 585 [8] settles the matter, stating:

“The generic parameters are not preserved in instances created with parameterized types, in other words generic types erase type parameters during object creation.”

In practice, Python implementations *ignore* type erasure and keep records of generic types in run-time objects. Listing 3 demonstrates how to obtain information about type parameters and generic arguments at run time.



■ **Figure 2** Crashing the Pyre playground with an infinite subtyping cycle.

■ **Listing 3** Obtaining reified generic types in Python at run time.

```
from typing import TypeVar, Generic
x = TypeVar("x", contravariant=True)
class C(Generic[x]): ...
print(C.__parameters__) # (-x,)
y = C[int]()
print(y.__orig_class__) # __main__.C[int]
```

The field `__parameters__` of the generic class `C` reveals that the class uses a single contravariant type parameter `x`. The generic type of the parameterized instantiation `C[int]()` is retrieved using the `__orig_class__` field. We tested the code in Listing 3 on several Python distributions. The results are presented in Table 2.

■ **Table 2** Type reification in different Python distributions.

Python Distribution	Version	Type Reification	Notes
CPython	3.9.13	●	
PyPy	7.3.11	●	
RustPython	3.5.0alpha	●	
WinPython	3.0.20	●	
Brython	3.11.1	⊘	<code>__orig_class__</code> is bugged

Table 2 shows that all the Python implementations tested practice type reification. As an exception, Brython fails to retrieve the field `__orig_class__`. After inspecting the source code, the author believes it to be a bug. At the moment of writing these lines, an open issue¹ on CPython's GitHub project calls for a proper API wrapper for the `__orig_class__` field. This development indicates that the Python community sees type reification not as a mere implementation detail but as a desired feature that should be established further.

Although Python reifies generic types, it does not mean that run-time type checkers use them in subtyping checks. And while static analyzers have all the type hints available, there is no guarantee that they implement the complete Python type system described in

¹ <https://github.com/python/cpython/issues/101688>

PEP 484. Whether or not a type checker gets stuck on infinite subtyping or successfully runs a subtyping machine is determined by its support of type parameter variance. In particular, both cases depend on contravariant type parameters. We reviewed various static and dynamic Python type checkers and checked whether they support variance as described in PEP 484. The results are presented in Table 3.

■ **Table 3** Variance support in Python type checkers.

Type Checker	Version	Typing Discipline	Variance Support	Notes
Mypy	0.991	static	●	
Pyre	0.9.17	static	●	
Pyright	1.1.279	static	⦿	Pyright is unsound
Pytype	2022.11.10	static	○	
Pyanalyze	0.8.0	static	○	
Pydantic	1.10.2	dynamic	○	
Typeguard	2.13.3	dynamic	○	
Pytypes	1.0b10	dynamic	○	Delegates to <code>isinstance</code> (sometimes)
Typical	2.8.0	dynamic	○	Delegates to <code>isinstance</code>

We found that Mypy and Pyre are the only type checkers to fully support subtyping with variance. Pyright also acknowledges variant type parameters, but reports errors for correctly-typed programs, meaning that it is unsound. We showed this by running one of our (accepting) subtyping machines: Mypy and Pyre reported no errors while Pyright did report one. The code is found in the supplementary material. Pytype reported that it does not support contravariant parameters. For the rest of the type checkers, we had to search the source code for any mention of variance to conclude that they do not support it. Typical seems to delegate subtyping checks to `isinstance` or equivalent methods that reject generic types altogether. We observed similar behavior in Pytypes when using type forward references, i.e., types in string literals.

To fully support PEP 484, Python type checkers must support covariant and contravariant type parameters. Variance, however, introduces a security vulnerability in the form of unavoidable infinite subtyping cycles. Crashes of static type checkers may be forgiven, but for dynamic checkers, infinite subtyping poses a major concern. This also holds for Python’s `isinstance` check, if the designers of Python or one of its implementations ever consider adding run-time subtyping checks against generic types.

4 Real-Time Subtyping Machines

Grigore’s subtyping machines must scan the entire tape memory before they can simulate a single TM transition. This is because the subtyping machines can change their direction (from \blacktriangleleft to \blacktriangleright and vice versa) only when reaching the end of the tape. We now present an alternative design for subtyping machines that simulate TMs in real time, i.e., where a single TM computation step is simulated by $O(1)$ subtyping deductions.

Let $M = \langle Q, \Sigma, q_I, q_h, \delta \rangle$ be a TM where Q is the set of machine states, Σ is the set of tape symbols, q_I is the initial state, q_h is the termination state, and

$$\delta : Q \times (\Sigma \cup \{\perp\}) \rightarrow Q \times \Sigma \times \{L, R\}$$

is the transition function. In each computation step, the TM changes its state, overwrites the current tape cell, and moves the machine head to the left (L) or right (R). The TM accepts its input if and only if it reaches the termination state q_h . We use the symbol $\perp \notin \Sigma$ to denote blank tape cells.

44:10 Python Type Hints Are Turing Complete

The new encoding of subtyping machines is shown in Table 4. The encoding comprises ten inheritance rules (i) to (x) (recall that we use a colon to denote inheritance). To encode a TM M as a subtyping machine, fill in the inheritance rules' missing values using elements from M that satisfy the conditions on the right-hand side. For example, if M contains state $q_4 \in Q$ and tape symbol $p \in \Sigma$, then by rule (v),

$$Q_4^{LL}x : L_p N Q_4^L L_p N x.$$

When multiple rules apply to the same type, multiple inheritance is used. Symbol x is a contravariant type parameter and symbol $?$ denotes the wildcard type, i.e., the type that is consistent with (can be substituted by) any type. In Python, this is the `Any` type [16]. All the type parameters used in the encoding are contravariant. There are ten more inheritance rules in addition to those in Table 4, obtained by swapping L and R in each rule in the table.

■ **Table 4** Real-time subtyping machines. For each inheritance rule, swap L and R to get the symmetrical rule. The type parameter x is contravariant.

(i)	$Q_s^L x : L_a N Q_s^L L_b N x$	$\delta(q_s, a) = \langle q_{s'}, b, L \rangle$
(ii)	$Q_s^L x : L_a Q_s^{LRR} N L_b N x$	$\delta(q_s, a) = \langle q_{s'}, b, R \rangle$
(iii)	$Q_s^L x : L_{\perp} Q_s^{L\perp L} N L_b N x$	$\delta(q_s, \perp) = \langle q_{s'}, b, L \rangle$
(iv)	$Q_s^L x : L_{\perp} Q_s^{L\perp R} N L_b N x$	$\delta(q_s, \perp) = \langle q_{s'}, b, R \rangle$
(v)	$Q_s^{LL} x : L_a N Q_s^L L_a N x$	$\forall q_s \in Q, \forall a \in \Sigma$
(vi)	$N x : Q_s^{LRR} N Q_s^{RR} x$	$\forall q_s \in Q$
(vii)	$N x : Q_s^{L\perp L} Q_s^{RL} N L_{\perp} N x$	$\forall q_s \in Q$
(viii)	$N x : Q_s^{L\perp R} N Q_s^{RR} L_{\perp} N x$	$\forall q_s \in Q$
(ix)	$N x : Q_s^{LR} N Q_s^R x$	$\forall q_s \in Q$
(x)	$Q_h^L x : L_a ?$	$\forall a \in \Sigma \cup \{\perp\}$

The roles of the types in Table 4 are mostly the same as in Grigore's encoding (Table 1), i.e., L_a is a tape cell containing the symbol a , N is a buffer type, and Q_s is a state type. The new encoding, however, does not use a type for the machine head (M in Grigore's encoding) because the state type Q also indicates the location of the machine head. The superscripts of Q imply the head's movement direction; e.g., Q_s^{LL} means that the head is about to go two cells to the left, $Q_s^{L\perp R}$ means that the head is about to move left into a blank cell and then rotate, and so on.

The initial TM configuration is encoded by the following subtyping query:

$$Z N L_{\perp} Q_I^R \blacktriangleleft L_{a_1} N L_{a_2} N \cdots L_{a_{m-1}} N L_{a_m} N L_{\perp} N Z \quad (9)$$

The content of the tape is encoded by the L types, read from the left to the right. The current state and the position of the machine head are encoded by the type Q_I^R – the current cell is on the right (R) of the type, which is also the direction of the query (\blacktriangleleft). The infinite blank ends of the tape are encoded by the type L_{\perp} . Observe that the colors of the types in Equation (9) match the colors of the corresponding TM components in Figure 1. Type Q_s^R is half red and half orange because it represents both the current state *and* the head location.

To prove the correctness of the simulation, we show that the subtyping query in Equation (9) simulates the TM transitions while preserving the encoding of the machine tape, head, and state comprising the TM configuration. There are three variables to be considered:

the initial orientation of the head, whether or not the current cell is blank, and whether or not the machine head changes direction. For example, the machine head Q_I^R in Equation (9) points to the right (R) and reads a non-blank cell L_{a_1} . The next cell could be either L_{a_2} , if the head continues right, or blank L_\perp , if it rotates.

We now cover four cases, assuming that the initial orientation is *left* (this is the orientation used in Table 4). The other four cases are symmetrical. Next to each subtyping deduction step, we mention the subtyping rule used in this step. Recall that there are two subtyping rules, SUPER and VAR (Equations (6) and (7)). As all the type parameters are contravariant, the query changes direction (from \blacktriangleleft to \blacktriangleright and vice versa) after applying VAR a single time.

Case I. The head points to a *non-blank* cell a , replaces it with symbol b , and continues *left*. The relevant TM transition is

$$\delta(q_s, a) = \langle q_{s'}, b, L \rangle$$

and the resulting subtyping query is

$$\begin{aligned} \dots NL_a \blacktriangleright Q_s^L \dots & \\ \dots NL_a \blacktriangleright L_a N Q_{s'}^L L_b N \dots & \quad (i) + (\text{SUPER}) \\ \dots \blacktriangleright Q_{s'}^L L_b N \dots & \quad (\text{VAR}) \times 2 \end{aligned}$$

“(i) + SUPER” means applying rule SUPER with inheritance rule (i) from Table 4. “(VAR) $\times 2$ ” means applying rule VAR twice. Observe that the subtyping query simulates the TM transition while preserving the encoding: symbol L_a is replaced by L_b , state Q_s is replaced by $Q_{s'}$ (the next machine state), and the head moves to the cell on the left.

Case II. The head points to a *non-blank* cell a , replaces it with symbol b , and continues *right*. The relevant TM transition is

$$\delta(q_s, a) = \langle q_{s'}, b, R \rangle$$

and the resulting subtyping query is

$$\begin{aligned} \dots NL_a \blacktriangleright Q_s^L \dots & \\ \dots NL_a \blacktriangleright L_a Q_{s'}^{LRR} NL_b N \dots & \quad (ii) + (\text{SUPER}) \\ \dots N \blacktriangleleft Q_{s'}^{LRR} NL_b N \dots & \quad (\text{VAR}) \\ \dots Q_{s'}^{RR} N Q_{s'}^{LRR} \blacktriangleleft Q_{s'}^{LRR} NL_b N \dots & \quad (vi) + (\text{SUPER}) \\ \dots Q_{s'}^{RR} \blacktriangleleft L_b N \dots & \quad (\text{VAR}) \times 2 \\ \dots NL_b Q_{s'}^R NL_b \blacktriangleleft L_b N \dots & \quad (v) + (\text{SUPER}) \\ \dots NL_b Q_{s'}^R \blacktriangleleft \dots & \quad (\text{VAR}) \times 2 \end{aligned}$$

Case III. The head points to a *blank* cell (the end of the tape), replaces it with symbol b , and continues *left*. The relevant TM transition is

$$\delta(q_s, \perp) = \langle q_{s'}, b, L \rangle$$

and the resulting subtyping query is

$$\begin{aligned} ZNL_\perp \blacktriangleright Q_s^L \dots & \\ ZNL_\perp \blacktriangleright L_\perp Q_{s'}^{L\perp L} NL_b N \dots & \quad (iii) + (\text{SUPER}) \\ ZN \blacktriangleleft Q_{s'}^{L\perp L} NL_b N \dots & \quad (\text{VAR}) \\ ZNL_\perp N Q_{s'}^{RL} Q_{s'}^{L\perp L} \blacktriangleleft Q_{s'}^{L\perp L} NL_b N \dots & \quad (vii) + (\text{SUPER}) \\ ZNL_\perp N Q_{s'}^{RL} \blacktriangleright NL_b N \dots & \quad (\text{VAR}) \\ ZNL_\perp N Q_{s'}^{RL} \blacktriangleright Q_{s'}^{RL} N Q_{s'}^L L_b N \dots & \quad (ix) + (\text{SUPER}) \\ ZNL_\perp \blacktriangleright Q_{s'}^L L_b N \dots & \quad (\text{VAR}) \times 2 \end{aligned}$$

44:12 Python Type Hints Are Turing Complete

Case IV. The head points to a *blank* cell, replaces it with symbol b , and continues *right*. The relevant TM transition is

$$\delta(q_s, \perp) = \langle q_{s'}, b, R \rangle$$

and the resulting subtyping query is

$$\begin{aligned} ZNL_{\perp} &\blacktriangleright Q_s^L \dots && \\ ZNL_{\perp} &\blacktriangleright L_{\perp} Q_{s'}^{L\perp R} NL_b N \dots && (iv) + (\text{SUPER}) \\ ZN &\blacktriangleleft Q_s^{L\perp R} NL_b N \dots && (\text{VAR}) \\ ZNL_{\perp} Q_{s'}^{RR} N Q_{s'}^{L\perp R} &\blacktriangleleft Q_{s'}^{L\perp R} NL_b N \dots && (viii) + (\text{SUPER}) \\ ZNL_{\perp} Q_{s'}^{RR} &\blacktriangleleft L_b N \dots && (\text{VAR}) \times 2 \\ ZNL_{\perp} NL_b Q_{s'}^R NL_b &\blacktriangleleft L_b N \dots && (v) + (\text{SUPER}) \\ ZNL_{\perp} NL_b Q_{s'}^R &\blacktriangleleft \dots && (\text{VAR}) \times 2 \end{aligned}$$

The TM rejects its input when its current state is q_s , the current tape symbol is a , and the transition $\delta(q_s, a)$ is not defined. In this case, the subtyping query $L_a \blacktriangleright Q_s^L$ also rejects since there is no inheritance rule in Table 4 with which rule SUPER can be applied. On the other hand, if the TM reaches state q_h and accepts its input, the subtyping query $L_a \blacktriangleright Q_h^L$ is resolved by applying rule (x) .

Note that our simulation is clearly real-time. To simulate a single TM transition, the subtyping machine performs at most eight subtyping deductions (in cases II and IV).

The wildcard type used in rule (x) is not a part of Kennedy and Pierce's system of nominal subtyping with variance. Instead of using the wildcard, the subtyping machine could go to either side of the tape before resolving the query, as done in Grigore's simulation. This makes the subtyping machine design a bit more complicated, and its simulation of the TM returns to be non-real-time, but the computational complexity of the simulation is not increased.

5 Implementation and Performance Experiment

We present our Python implementation of Grigore's original reduction and our new real-time simulation introduced in Section 4 in the supplementary material. Our implementation compiles TMs into Python subtyping machines that use the type hints and generics described in PEP 484 [16]. Each subtyping machine comprises a class table (Table 4) and a variable assignment that invokes a subtyping query (Equation (9)). To run the subtyping machine, we use Mypy to type-check the generated Python code.

If the subtyping machine accepts its input, Mypy terminates successfully, and if the input is rejected, Mypy reports a typing error. But what happens when the subtyping machine runs indefinitely? When running Mypy on the code in Listing 2, containing an infinite subtyping cycle, Mypy crashes with a segmentation fault. To uncover the reason for the segmentation fault, we remove a call to `sys.setrecursionlimit` from Mypy's source code and run it again with the flag `--show-traceback`. Mypy reports the following error:

```
...
File "mypy/types.py", line 1283, in accept
File "mypy/subtypes.py", line 585, in visit_instance
File "mypy/subtypes.py", line 345, in check_type_parameter
File "mypy/subtypes.py", line 339, in check
File "mypy/subtypes.py", line 179, in is_subtype
File "mypy/subtypes.py", line 329, in _is_subtype
```

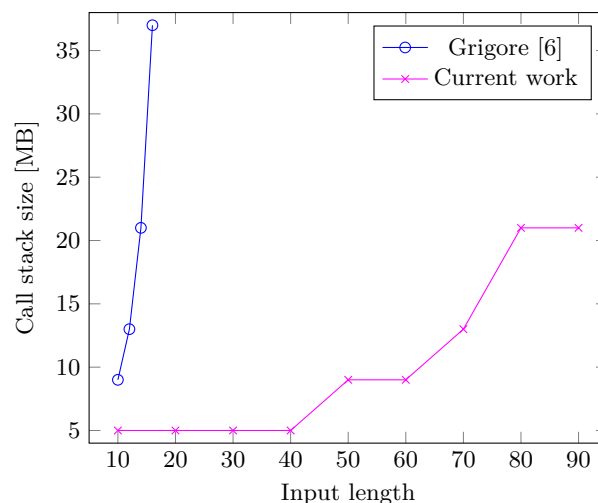
```

File "mypy/types.py", line 1283, in accept
File "mypy/subtypes.py", line 585, in visit_instance
File "mypy/subtypes.py", line 345, in check_type_parameter
File "mypy/subtypes.py", line 339, in check
File "mypy/subtypes.py", line 147, in is_subtype
RecursionError: maximum recursion depth exceeded while calling a Python object

```

The stack trace confirms that Mypy’s subtyping algorithm is implemented using recursion, causing it to crash with a segmentation fault on infinite subtyping. This observation makes it possible to measure the run time of the subtyping machine, i.e., the number of subtyping deductions it performs, by calculating the minimal size of the call stack required for Mypy to type-check the machine.

Figure 3 describes the results of our experiment, in which we measured the run times of subtyping machines accepting input words of various lengths. The TM used in the experiment recognizes palindromes over $\{a, b\}$ and runs in $O(n^2)$. We compiled the TM together with random palindromes of increasing lengths into subtyping machines, once using Grigore’s method and once with our construction. Then, binary search was used to find the minimal call stack size (in megabytes (MB)) required for Mypy to type-check the machine without getting a segmentation fault.



■ **Figure 3** Run times of the palindrome subtyping machines: Grigore’s reduction vs. the new reduction.

In theory, Grigore’s subtyping machines should run in $O(n^3)$ due to their inherent slow-down, while our machines are expected to run in $O(n^2)$ since they simulate the palindromes TM in real time. In practice, we see that our subtyping machines are much faster than Grigore’s and require significantly fewer resources to be type-checked.

6 Conclusions

Python type hints are Turing complete because PEP 484 supports nominal subtyping with variance, including contravariance and expansive-recursive inheritance. These two subtyping features are sufficient to implement Grigore’s subtyping machines that simulate TMs at compile time.

We demonstrated that infinite subtyping cause Python type checkers to crash with a stack overflow error. Due to the undecidability of the halting problem, fixing this problem requires changing the Python type system described in PEP 484. Even run-time type checkers are in danger because existing Python distributions reify generic types – in spite of the language specifications. In practice, we found that only two static analyzers (Mypy and Pyre) are vulnerable to infinite subtyping since they provide the most complete implementations of Python’s type system.

We described an alternative subtyping machine design that simulates TMs in real time, removing the inherent slowdown introduced in Grigore’s original design. Our experiment shows that the new subtyping machines compile significantly faster. Our design is an essential step towards practical subtyping machine applications. Nevertheless, such applications would most likely depend on the type checker’s implementation of the subtyping algorithm – specifically, that it is not recursive.

References

- 1 ECMA International. *ECMA Standard 335: Common Language Infrastructure*, 3 edition, June 2005. Available at <http://www.ecma-international.org/publications/standards/Ecma-335.htm> (accessed Aug. 2022).
- 2 Martin Fowler. *Fluentinterface*, 2005. URL: <https://www.martinfowler.com/bliki/FluentInterface.html>.
- 3 Yossi Gil and Tomer Levy. Formal language recognition with the Java type checker. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th Europ. Conf OO Prog. (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Inf. (LIPIcs)*, pages 10:1–10:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2016.10.
- 4 Yossi Gil and Ori Roth. Fling – A fluent API generator. In Alastair F. Donaldson, editor, *33rd Europ. Conf OO Prog. (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Inf. (LIPIcs)*, pages 13:1–13:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2019.13.
- 5 Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting f-bounded polymorphism into shape. *SIGPLAN Not.*, 49(6):89–99, June 2014. doi:10.1145/2666356.2594308.
- 6 Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009871.
- 7 Andrew Kennedy and Benjamin Pierce. On decidability of nominal subtyping with variance. In *Int. Work. Found. & Devel. OO Lang.*, FOOL/WOOD’07, Nice, France, January 2007. ACM. URL: <http://foolwood07.cs.uchicago.edu/program/kennedy-abstract.html>.
- 8 Lukasz Langa. *PEP 585 – Type Hinting Generics In Standard Collections*. Available at <https://peps.python.org/pep-0585/> (accessed Nov. 2022).
- 9 Jukka Lehtosalo, Guido van Rossum, Ivan Levkivskiy, and Michael J. Sullivan. *mypy*. Available at <http://mypy-lang.org/> (accessed Aug. 2022).
- 10 Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: a fluent API generator. In *Proc. 16th ACM SIGPLAN Int. Conf Generative Prog.*, GPCE’17, pages 199–211, Vancouver, BC, Canada, 2017. ACM.
- 11 Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. *Scala language specification version 2.13*. Available on: <https://scala-lang.org/files/archive/spec/2.13/> (accessed Aug. 2022).
- 12 Oracle. *The Java Language Specification, Java SE 8 Edition*, February 2015. Available at <https://docs.oracle.com/javase/specs/>.

- 13 Emil Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.
- 14 Ori Roth. Study of the subtyping machine of nominal subtyping with variance. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485514.
- 15 Ross Tate, Alan Leung, and Sorin Lerner. Taming wildcards in java’s type system. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’11*, pages 614–627, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993570.
- 16 Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. *PEP 484 – Type Hints*. Available at <https://peps.python.org/pep-0484/> (accessed Nov. 2022).
- 17 Guido van Rossum and Ivan Levkivskyi. *PEP 483 – The Theory of Type Hints*. Available at <https://peps.python.org/pep-0483/> (accessed Nov. 2022).
- 18 Todd Veldhuizen. *Using C++ Template Metaprograms*, pages 459–473. SIGS Publications, Inc., USA, 1996.
- 19 Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. Generating a fluent api with syntax checking from an LR grammar. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360560.

